REAL TIME GRAPHIC STATUS DISPLAY OF C-DOT DIGITAL SWITCHING SYSTEM

(Graphic Interface Module)

ARUN VISWANATHAN

School of Computer and System Sciences
Jawaharlal Nehru University
New Delhi
May, 1989

REAL TIME GRAPHIC STATUS DISPLAY OF C-DOT DIGITAL SWITCHING SYSTEM

(Graphic Interface Module)

Dissertation submitted to Jawaharlal Nehru University in partial fulfillment of requirements for the award of the degree of

MASTER OF TECHNOLOGY

in

Computer Science and Technology

by

ARUN VISWANATHAN

School of Computer and System Sciences
Jawaharlal Nehru University
New Delhi

May, 1989

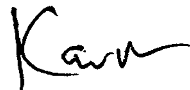
to my parents

Certificate

This work titled: **REAL TIME GRAPHIC STATUS DISPLAY OF C-DOT DIGITAL SWITCHING SYSTEM** (Graphic Interface Module) has been carried out by Mr. Arun Viswanathan, a bonafide student of School of Computer and System Sciences, Jawaharlal Nehru University.

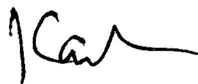
This work is original and has not been submitted so far in part or full for any degree or diploma in any other University or Institute.

ARUN VISWANATHAN
Candidate



Prof. Karmeshu
Supervisor
SCSS, JNU, New Delhi.

Prof. B. S. Khurana
Supervisor
SCSS, JNU, New Delhi.



Prof. Karmeshu
Dean,
SCSS, JNU, New Delhi.

Contents

Preface vii
Abstract x

Chapter 1 **1**

Project Overview 1
1.1 Configuration 2
1.2 The Libraries 3
1.3 Application Layer 4
1.4 Development Environment 4

Chapter 2 **6**

The SOAP Library 6
2.1 The SOAP Protocol 7
2.2 Design Consideration 14
2.3 The PC End 18
2.4 The IOP End 19
2.5 SOAP Library routines 25

Chapter 3 **30**

The CGRAPHIC Library 30

3.1 Library structure 31
3.2 Core Routines 33
3.3 Geometric Routines 34
3.4 String Manipulation Routines 39
3.5 Cursor Control Routines 42
3.6 Mapping Routines 44
3.7 Miscellaneous Routines 45
3.8 Debugging Aids 47

Chapter 4 **49**

The GIM 49
4.1 Hardware Requirements 50
4.2 Software Architecture 50
4.3 Alarm Display 58
4.4 Error Messages 58
4.5 Duplex Mode 58

Chapter 5 **60**

Conclusion 60
5.1 Protocol 60
5.2 Graphics 61
5.3 Application Layer 62

Appendix A **64**

Glossary 64

Appendix B **67**

Notes on C-DOT DSS 67
B.1 Hardware Architecture 67
B.2 Software Architecture Overview 69
B.3 Development Environment 72

Appendix C 73

Notes on Serial Communication 73

Appendix D 76

Notes on Interrupts 76
D.1 Types of interrupts 76
D.2 Hardware Interrupts 77

Appendix E 79

Notes on the IBM EGA 79
E.1 Display Memory Organization 80
E.2 Bit Mask Register 81
E.3 Map Mask Register 81
E.4 Set/Reset Register 81
E.5 EGA Write Modes 82
E.6 EGA Color Palettes 82
E.7 Data Rotate Register 83

Appendix F 84

System And Function Calls 84
F.1 UNIX System Calls 84
F.2 'C' Function Calls 85
F.3 C-ISAM Function Calls 86

Appendix G 88

Bibliography 88

Appendix H 91

Source Listing 91
H.1 CGRAPHIC Listing 91
H.2 SOAP Listing 95
H.3 GIM Listing 99
H.4 Snapshots 101

Preface

Real time systems have varied applications such as, Petrochemical plants, Nuclear plants, Digital Switching systems etc. All these systems have one thing in common for which they are called real time systems: they are required to cater to the external environment imposing time constraints to be met at all cost. These time constraints are usually of the order of few seconds or even less. Due to these constraints the system is required to keep pace with the changes and demands of the external world with which they interact. Moreover they pose stringent requirements on response time. Summing up, real time systems are very sensitive to the external world and are mercurial in their status - "A real time system is that which responds spontaneously to extraneous events or inputs".

Hence it is essential that some kind of monitoring panel exist along with these systems so that their activity can be closely observed. This manuscript discusses the design and implementation of such a software driven display panel developed for the Center for Development of Telematics' (C-DOT) Digital Switching System (DSS).

Since the real time systems have always evinced interest in me, I approached C-DOT for a project. Their acceptance in addition to the motivation and encouragement rendered by my supervisors Prof. Karmeshu and Prof. B. S. Khurana made me sign up with the project Graphic Interface Module (GIM). I was assigned to the Operating Systems Group (OS Group) headed by Mr. H. Ghosh. The project was to graphically display the status and performance of the C-DOT DSS on real time basis. This project has been successfully completed and the

package is running at the C-DOT's Exchange at Ulsoor, Bangalore.

Constitution of the manuscript

This work has been divided into five chapters. Chapter 1 is basically a conspectus of the total project. But before going through any of the chapters it might be useful to riffle through Appendix B which gives an overview of the C-DOT's DSS software and hardware Architecture. Chapter 2 deals with the protocol library SOAP.[1] It considers the conceptual and implementation aspects of the protocol. Related to this chapter we have two Appendices C and D. They give a brief note on the Serial Communication and Interrupts respectively. Chapter 3 deliberates on the graphic library CGRAPHIC developed for the IBM Enhanced Graphic Adapter (EGA). This chapter expands on the functions available with this luxuriant library. Also a brief note on EGA is available in Appendix E. Chapter 4 deals with the GIM package. It describes the design and implementation of the package. Chapter 5 takes a retrospective view of the project and points out the possible enhancements and improvements that could be inoculated. Appendices A, F, G and H contain Glossary, Unix System calls, Bibliography and the source listing respectively. Most of this work is self-contained and requires not much of pre-requistes since sufficient information has been supplied in the appendices.

Acknowledgments

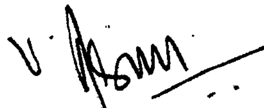
I wish to express my sincere thanks and deep sense of gratitude to my supervisors Prof. Karmeshu and Prof. B. S. Khurana for their keen interest, valuable guidance, inspiration, and constructive criticism during the course of this work and particularly through the planning and completion of this manuscript.

I am more than grateful to H. Ghosh, Group Leader, OS Group, C- DOT, under whose dynamic guidance and close monitoring I have been able to successfully complete this project. His consummate knowledge and skills in Computer Science was a great source of learning and help. My gratitude to him goes beyond what is expressed herein. I am equally

1 Simplest Of All Protocols.

grateful to S. Ramanathan. It was a fruitful experience to have worked with him. There was a lot to learn from his inordinately mellifluous 'C' code. He was always there to help me out of the drudgery of finding elusive bugs. Special thanks are due to K. Pramod and Ronki Shariwale for critically analyzing this manuscript and suggesting many useful and meaningful changes. Thanks are also due to S. Sreedhar, Ashis Chattopadhyaya and Yogesh Sharma for providing a very amicable work atmosphere and also for the secours rendered by them. Finally, I would like to the thank C-DOT for providing all the facilities and resources to make this project a success. In all, experiences in C-DOT will ever remain to be memorable and cherishable.

I would also like to extend my thanks to my colleagues Sunil Sindwani and Naveen Jain for all the help they rendered.



Arun Viswanathan

May 1989, New Delhi.

Abstract

In any large and complex real time system such as Thermal Power stations, Power distribution units, Nuclear Reactors, Telephone switching systems, Chemical factories etc., it is essential that status of the entire system be be shown on a display panel. The task of analyzing the status of these systems, which involve innumerable parameters, can be reduced to a great extent if the software driven display panel shows the statuses graphically by means of flow diagrams, charts, etc. Moreover, the tedium involved in system performance evaluation can be reduced almost to zero. Such a display panel can also help in taking corrective measures at the appropriate time.

Since real time systems are very sensitive their statuses can change in a matter of seconds or even less. Therefore, up-to-date status and performance related data should be available to such display panels for presentation and moreover they are required to work off-line without loading or hindering the system's functioning. Hence a reliable data transfer mechanism should be established to transfer data from the system to the off-line computer (display panel). Furthermore, this display unit should give fast response to the user: that is the status and performance screens should be made as fast as possible, using the latest modified data. This calls for an enhanced graphic facility to draw screens fast and thereby reducing the response time. Also the protocol for data transfer should be fast and reliable.

This manuscript deliberates on the design and implementation of such a package for the C-DOT's (Center for Development of Telematics) Digital Switching System (DSS).

The Graphic Interface Module is a graphic package that will fetch data related to the status and performance of the C-DOT DSS MAX (Main Automatic Exchange) from a UNIX machine through a protocol and show it graphically on a video screen. The GIM is a totally

menu-driven package. This has been built as an application layer over two libraries developed for this purpose, which otherwise are fully generalized. One library has routines related to the protocol for data transfer from the host machine to the off-line computer. The other one is a graphic library. The Graphic Interface Module (GIM) has been implemented on an IBM/AT with an EGA card. This package has been developed and has also been tested out at the C-DOT's Exchange at Ulsoor, Bangalore and is working satisfactorily.

Chapter 1

Project Overview

In any large organization, such as a thermal power plant, a petrochemical complex, or a nuclear reactor etc., it is customary to have a large control room in which the status of various parameters of processes or machines is continuously displayed as alarm annunciation or a line flow chart or video screens. When the number of parameters become very large, the size of the control room become larger and one operator cannot manage to keep an eye on the whole process. With the advent of computers, it is now possible to collect and store all parameters on an on-line basis, show generated alarms if required, and display the status on a Video Display Unit (VDU) in graphical manner to the minutest possible detail. The aim of this project is to design and implement such a display system which can show the status and performance related parameters of the C-DOT Digital Switching System (DSS) in a graphical manner on a real time basis, so that the manager or the operator of the switching system can know the status and performance of the Main Automatic Exchange (MAX) at a glance. This will not only make the MAX operator's work simple but also the handling of the system much easier. It will be easy for the operator to take necessary corrective measures related to the configuration and performance of the system. It can also facilitate the operator in localizing the faults to various modules/sub-modules/units. Even the tedium involved in analyzing exchange performance can be reduced to a great extent. Finally, but not the least, such a package can increase the salability and marketability of the C-DOT switch.

1.1 Configuration

The data base related to the status and performance of the MAX is stored in the Input Output Processor (IOP), which supports a UNIX environment. This is a full fledged computer based on the Motorola 68000 processor. For detailed information on the C-DOT DSS architecture the reader is referred to Appendix B, [6]-[8], and [23]-[25]. This data base gets periodically updated as and when the statuses are changed by the processes in the module where the actual call processing related activity takes place. The objective of the project is to access this data and show it in a visual form. Earlier such a configuration was available with a Tektronix Graphic terminal attached to the IOP. But that had a severe disadvantage; it used to load the IOP for its computations. So it was decided to attach an IBM PC XT/AT to the IOP, which is quite powerful and provides good graphic facilities. The advantage of such a configuration is that we can have data fetched on demand, thus reducing the traffic between the IBM PC XT/AT and the IOP, and all the processing can be offloaded to the IBM PC XT/AT while keeping the activity at the IOP minimal. Moreover, the IBM PC XT/AT could be attached to both the IOPs (Master and Slave) [cf. Appendix B] through two of its four serial communication (COM) ports. The IBM PC XT/AT would communicate with one of the IOPs and would automatically switch to the other IOP when it is not able to communicate with the first one. Such a duplex configuration would provide uninterrupted display of status and performance related parameters.

The configuration stated above engender the availability of two basic building blocks - firstly, a protocol for transfer of data from the IOP to the IBM PC XT/AT via the RS-232 serial hardware interface. The protocol will increase the data reliability and can moreover reduce data redundancy. Another advantage of such a protocol is that, we can transfer only the desired number of bytes from the IOP, thus making the work simple and fast as required by real time environment. Secondly, a good graphics library that provided fast and useful routines. For this reason we had chosen an IBM PC XT/AT (the reader may note that the IBM PC XT/AT will hereafter be referred to as PC) with Enhanced Graphics Adapter (EGA). We have not used any of the MS-DOS internals in either of the basic building blocks. So these routines, though developed in MS-DOS environment, could be ported to a machine having a different environment with only minor changes. These two basic building blocks

have been developed as libraries. The SOAP[1] Library provide routines to communicate with the IOP. The CGRAPHIC Library provide routines to draw geometric figures, string manipulation, etc. in graphics mode (16/14) of the IBM EGA.

1.2 The Libraries

Though the two libraries have been built as substratum to facilitate the development of GIM (Graphic Interface Module), they are totally general and completely extendible. The SOAP Library routines have been built over a lower level protocol. In fact, the 'handshaking' principle is carried out by this lower level protocol. Towards this end a whole driver has been developed at the PC-end and an application layer over the IOP driver to serve the request commands from the PC. At the PC-end the serial port driver has been built as Interrupt Service Routines (ISR). These interrupts use the serial COM ports of the PC for their communication with the IOPs. At the IOP-end the application program interprets the request and serves it. This library and its counterpart in IOP have been discussed extensively in Chapter 2. All the library routines have been developed through register programming. The serial port controller is directly manipulated to achieve transfers.

The CGRAPHIC Library routines directly manipulate the IBM EGA registers and access the EGA display memory. Almost all the routines in the library have a counterpart in the library that are lower level routines accessing the hardware directly. The lower level routines are also available to the application programer. Chapter 3 deals with this library.

We have tried to keep the SOAP Library as general as possible, while serving our specific requirements. Needless to say, the CGRAPHIC Library is totally generalized. Both the libraries are available with 'C' interface, i.e., the routines in the library can be directly called in any 'C' program. Both the libraries have been built keeping speed, as the primary consideration, in mind. They are neat, easy to use and fast. An added advantage of these libraries is that they can be enhanced by the user to suit his purpose by using the lower level routines that have been made available to him. We have also tried to maintain code optimization in these libraries but whenever a need for tradeoff was there, we preferred time over space. This was typically because these routines would be used in a real time

1. Simplest Of All Protocols.

environment as far as our requirements were concerned.

1.3 Application Layer

Using these two libraries a Menu-driven application layer has been built. The status of each module of the MAX is shown in a page. The user can also expand on any of the sub-modules being shown in that page. The application layer follows a forest structure. By default we show all the sibling pages of a particular tree of the forest in round robin loop. If the user wants he can go to any of the sibling page or expand on a child or go to the parent page. The user can also escape to the main- menu page from any where in the forest and hence can jump to the root of any tree in the forest. We also show the alarms that arise in the MAX along with its status. The application layer has been developed by using highly expedient data structures and we have used them freely. This has given modularity to the whole package. This approach also helped us in adding pages, changing configuration of the pages, etc. This modularity will not only help in maintaining the package but also in enhancing it when required. The application layer also provides the user with saving a displayed page onto a storage system or replay an already stored page and many other such facilities. This application layer, which is the major objective of the project, has been named Graphic Interface Module (GIM). Chapter 4 is devoted to GIM.

1.4 Development Environment

Since the source code is big, we feel that a comment should also go about it. A part of the source code is given in Appendix H. All of the programming is in 'C' programming language except for two routines in SOAP that are written in assembly language. We have followed the C-DOT coding guidelines in writing our programs. We have tried to maintain modularity at each phase of the project. This has been achieved through data abstraction, i.e., we have tried that routines do not become dependent on the data structures of the other routines they call. All variables and function names are in lower case letters. The functions that are hardware dependent start with an 'underscore'. All macros and hash-defines are in upper case letters. All typedefs start with an upper case letter. We have also tried to slip in lot of useful comments to help understand and maintain the package.

The working environment was an IBM/AT running MS_DOS with an EGA card with

256 KB of display memory and two serial COM ports. The other machine was a Motorola System 1000 running System V/68 (UNIX) with Motorola 68010 processor.

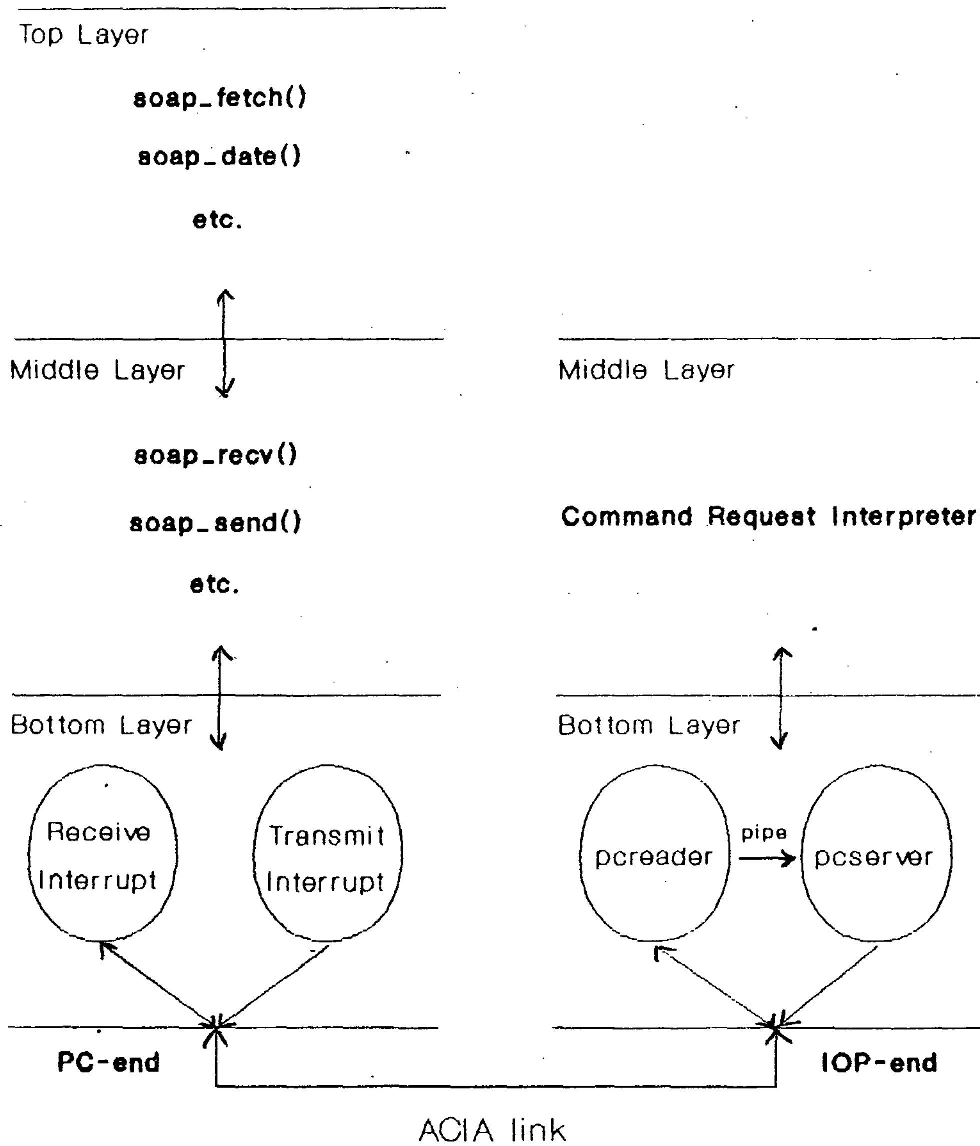
Chapter 2

The SOAP Library

The transfer of data from the IOP to the PC would require that the two machines establish some kind of communication protocol so that the reliability of the data transferred is of highest degree. Not only will the data reliability increase, but also data redundancy would reduce to almost zero. By data redundancy we mean that the same data being transferred more than once, even though data was transferred correctly, the first time. There could be another use of such a protocol; we can fetch only those many bytes that are required at an instant. This is of utmost importance to us, because the Graphic Interface Module (GIM) package showing real time status of the system would require that the response to the user be fast. One of the crucial factors determining this would be the amount of data transfer that we make. We wanted to reduce this to the barest minimum; that is, requesting for only those many bytes of data that would be needed to make up a page.

At first a lower level protocol was developed that would transmit and receive data packets, and validate them. Before transmission the lower level protocol would embed the data in a standard format that both sides recognize. The lower level protocol at the PC-end has been implemented as interrupts. Some of the lower level routines of the SOAP Library have been developed over these interrupts. The standard SOAP Library routines have been built by using these lower level routines. Hence at the PC-end there exists three layers of the protocol. That is,

- top layer: standard SOAP routines
- middle layer: lower level routines



SOAP Software architecture

Figure 2.1

- bottom layer: interrupts.

The routines of subsequent two layers are also available to the user. He can use them to enhance the SOAP Library. But it calls for deligence on the part of the user while using these lower level routines. The SOAP Library has a server at the IOP-end. This IOP-end counterpart has been developed as an application layer over the UNIX driver. There are two processes at the IOP-end. One reads the port, while the other forms the packets and transmits them to the PC. These two processes establish a 'pipe' for inter process communication.

As far as the present implementation at the IOP-end is concerned, it remains to be a server to the requests generating from PC-end. We have not developed any library or request facility at the IOP-end. This was because that such an implementation would be better if we were to write a whole driver at the IOP-end, otherwise the application can become quite slow. Moreover, the GIM project didn't have any such requirements. The IOP-end has a request interpreter that identifies the PC request and serves it. The figure 2.1 gives an overview of the whole communication protocol between the two machines.

2.1 The SOAP Protocol

The SOAP Protocol is based on the Tektronix, Inc.'s ICOM40 communication protocol. As the name suggests we have tried to keep the lower level protocol as simple as possible. We have avoided incurring unnecessary overheads in implementing this protocol. This was because our requirement was to transfer data from the IOP-end to the PC-end. But presently, the protocol developed, supports much more facilities and moreover it can be enhanced by any application programmer.

This section considers the protocol at the conceptual level. We have chosen serial transmission since this mode is cheaper than the parallel transmission. "Serial communication is cheaper than parallel because it requires fewer data lines - as few as two for two-way communication. Also the asynchronous mode of transmission makes much less

demand on the hardware because there is no need for special hardware to maintain synchronism between transmitter and the receiver".[1] As the communication over the serial data link interface RS-232 is asynchronous, a proper handshaking is required between the two machines to avoid reading noise as data and to re-transmit lost/corrupted data. Such a handshaking would also help the sender in knowing whether the data packet had reached uncorrupted, so that he can transmit further packets.

Now assume that the sender wants to transmit a packet to the other end. For this the sender needs some kind of signaling to inform the receiver that the data packet is following and he is also required to supply the data attributes such as data length, etc. Such a signal is said to constitute a Header. But now the receiver is required to distinguish between the Header and the Data block. For this purpose the Header and Data blocks can start with a control byte, e.g. we can keep a byte <SOH> indicating the start of Header. Similarly, when the receiver gets a packet he is required to inform the sender about it. For this purpose he needs two blocks, one to 'acknowledge' the successful receipt of the packet and another to mark the 'negative acknowledgment'. As these blocks also need to be distinguished from the other ones, we can take a similar approach by choosing two bytes <ACK> and <NAK> to indicate the start of the Ack and Nak block respectively.

Now lets see what should constitute the Header, Ack, Nak, and Data blocks. The task is simple. The Header should consist of the <SOH> indicating that a Header follows, and information about the data, i.e., data length, sequence number, source, destination, and a checksum. The data length field helps the receiver in knowing the length of the data that is to follow. Source field represents the identity of the process that has sent the data packet. Similarly, the destination field represents the identity of the process to whom this data packet should be handed over at the receiving end. The checksum will help in knowing whether the Header block had reached uncorrupted. There remains another field in the Header block that needs explanation-the sequence number. This field plays a major role in the lower level protocol. This field will represent the last packet that has been successfully transmitted by

1. [1] p.453.

the sender. This has been taken up in greater detail later in this section.

<SOH><data_length><sequence_number><source><destination><check_sum>
Header Block
<ACK><sequence_number>
Ack Block
<NAK><expected_sequence_number><error_code>
Nak Block
<STX><data><ETX><check_sum>
Data Block

Figure 2.2(a)

<SOH><data_length><sequence_number><source><destination><data><check_sum>
Clubbed Data Block
<ACK><sequence_number>
Ack Block
<NAK><expected_sequence_number><error_code>
Nak Block

Figure 2.2(b)

The Ack block will consist of a control byte <ACK> to identify the Ack block another field should be the sequence number indicating the sequence number of the last packet successfully received, while the Nak block will have a control byte <NAK> indicating the start of a Nak block. Since the packet did not reach successfully another field is required to indicate the type of error, so the sender can take appropriate corrective measures. Similar to the Ack block the Nak block should also contain a sequence number, but this time indicating the expected sequence number, i.e., one more than the sequence number of the last successfully received packet. Lastly, the Data packet could be sandwiched between two control bytes <STX> and <ETX> to indicate the start and end of the Data block respectively. Hence all the possible transmission blocks are distinctively identified by the control bytes. One more control byte <EM> will be required to indicate end of transmission (EM - End of Medium). This control byte will be sent by the receiver on successful receipt of a Data block. This may be called as the Em block. In fact, the Ack block can also be used to acknowledge the successful receipt of data packet. Figure 2.2(a) describes the blocks that have been discussed above.

Both the parties will communicate under the protocol stipulated below.

- The sender transmits a Header block.
- The receiver sends an Ack block to indicate successful receipt of the Header.
- The sender transmits the Data block.
- The receiver sends an Em block on successful receipt of a Data block.

The protocol is not as simple as to be able to put in the above four lines, they only remain to represent a general way of transaction. It is highly likely that many complications might arise while transactions take place. Say, for example, that the Header gets lost/corrupted, or an Ack gets lost/corrupted, or the Em gets lost etc. How should the sender and receiver ends react to such imminent circumstances ? Are they capable of handling all such occurrences ? And so on. The protocol is required to detect such occurrences and take corrective and remedial measures. But there could be such freak cases that might never occur to our mind. For such cases the protocol should support some kind of built-in recovery strategy, so that the transactions start taking place after a momentary lapse in communication. Nevertheless this work is not all that imponderable as it looks to be. A proper step by step evaluation of the protocol can help in developing a very stable communication protocol. We describe below some of the cases that could arise and also the steps to be taken.

Message Tracking

At any time of transaction a whole message can get lost. Hence both the parties are required to keep track of the number of successfully transmitted and received messages. This is where the sequence number in the Header block comes into picture. The receiver end can compare the sequence number in the Header with his successfully received sequence number to judge whether the packets are coming in proper order. Both, sender and receiver, are required to increment the successfully transmitted and received sequence number respectively at their ends to maintain seemliness in transmission of packets. The sequence number should be rolled back from 127 (or 255 if the byte is unsigned) to 1. The sequence number 0 is reserved for special purpose. At any time during the transactions either party can reset to sequence number 0, and neither of them should consider the sequence number 0 to be out of order. This can serve two purpose. One is that to start a new set of transactions. Secondly, it can be effectively used as one of the recovery strategies. The sender can

signal his re-sequencing request in the Header block while the receiver can request for re-sequencing through a Nak block; both blocks having sequence number equal to zero.

Lost Em Syndrome

Assume that the sender successfully transmits a Data block. The receiver will send an Em to acknowledge the event. Assume that, for reasons unknown, the Em gets lost. The sender will time out and try to re-transmit the Header of that Data block, for which he will receive a Nak. If the sender's successfully transmitted sequence number is two less than that of the expected sequence number in the Nak block the sender can safely assume that the Data block had reached successfully. Such an understanding is essential otherwise it can lead to redundant data.

Time-outs

It may happen that either side waits for an event that might not occur. In such cases, the party concerned may have to time-out. Lets consider an example to explain this. It may happen that the sender transmits a packet and receives no response from the receiving end and he may then have to time-out. For such reasons each party is required to time the events. Time-outs could be to made occur for one of the following reasons.

- receiver waits for a complete packet
- sender waits for an Ack/Nak

It is essential that time-outs are tuned properly. Consider that the receiver is reading a packet and due to noise in the link only a section of the packet reaches him and he waits for the remaining packet. While at the same time the sender will wait for an Ack/Nak. If the sender times-out first, he will re-transmit the packet and the receiver might take it to be the remnants of the packet on which he was waiting. Hence it is essential that the receiver times-out first and then the sender. Proper time-out can help in synchronization and restart a surceased activity. But a word of caution, time-outs are highly environment dependent. The party at the receiver end should always wait for further activity after time-outs rather than sending a Nak, since the sender will retry anyway, on time-out.

Error Conditions

The entire responsibility of checking whether the packet was transmitted successfully lies with the receiving end. The receiver can signal an error condition to the sender, who will then re-transmit the packet. The error conditions can be treated in two ways. Firstly, the receiver can signal an error condition through a Nak block with the appropriate error code. Secondly, the receiver on detecting a communication error goes into a wait state. The sender then times-out and re-transmits the packet. The different types of communication errors that could arise are listed below.

- sequence error
- checksum error
- no <STX> at start of Data block
- parity error
- framing error
- overrun error
- illegal destination identity
- invalid state error
- no buffer at receiver end
- multiple errors

If communication error like 'overrun error' occur very frequently, then it might require to lower the Baud rate. The 'invalid state error' means that both the parties were in incompatible states, for example, one party was in SEND state and received a Data or Header block.

Till now we have considered that the Header and Data block to be as separate units. But after re-thinking and re-analyzing we realized that it would be better for our application if they can be clubbed-up into one packet. See Figure 2.2(b) for the data format. The reason for this is that our Data blocks are manageably small (SOAP packets can accommodate 128 bytes of data) [cf. section 2.2] and we can avoid a lot of overheads by combining them with the Header block. After every Header is transmitted, it has to be validated and also Acked/Naked. Validation would require computing check sum and then comparing them. A similar procedure has to be followed with the Data block. Since the Data block is small and Header block is already small a re-transmission of the clubbed packet will not increase the effective transaction time. But if the Data blocks are big it is advisable to keep Header and Data blocks as separate units. The algorithm related to the actual protocol that has been

implemented is given in Figure 2.3 in pseudo code. The reader will notice that there isn't much of a change in the protocol by clubbing up the Header and Data block. In fact, all the things we said before hold good here too. The actual implementation may vary a little from the above algorithm, but the basic theme remains the same. The next section deals with precisely this aspect.

```
procedure sender;
begin
    reset retry_count;
retry: retry_count : retry_count + 1;
    if ( retry_count > max_retry_count ) then
        return ERROR;
    send the clubbed data packet;
    while ( not ACK/NAK ) do
        nothing; (* drop noise and wait for ACK/NAK *)
    case ( control_byte )
    begin
    TIME-OUT:
        goto retry;
    NAK:
        case ( nak_error_code )
        begin
        SEQUENCE_ERROR:
            if (transmit_sequence_number + 1 = nak_sequence_number ) then
            begin
                advance transmit_sequence_number;
                return SUCCESS;
            end (* fi *)
            else
            begin
                reset receipt_sequence_number;
                reset transmit_sequence_number;
                goto retry;
            end; (* esle *)
        OTHERS:
            goto retry;
        end; (* esac *)
    ACK:
        if ( transmit_sequence_number = ack_sequence_number ) then
        begin
            increment transmit_sequence_number;
            return SUCCESS;
        end (* fi *)
        else
            goto retry;
    end; (* esac *)
end;
```

```
procedure receiver;
begin
    reset retry_count;
retry: retry_count : retry_count + 1;
    if ( retry_count > max_retry_count ) then
        return FAILURE;
    while ( not SOH ) do
        nothing; (* drop noise and wait for SOH *)
    read Header of the clubbed packet;
    get data_length from the packet;
    read data_length number of bytes;
    if ( hardware_error ) then
    begin
        send Nak block with proper error_code;
        goto retry;
    end; (* fi *)
    if ( time_out ) then
        goto retry;
    validate checksum;
    if ( checksum_error ) then
    begin
        send NAK with proper error_code;
        goto retry;
    end; (* fi *)
    get the sequence_number in packet;
    if ( sequence_number = 0 ) then
    begin
        reset receipt_sequence_number;
        reset transmit_sequence_number;
    end; (* fi *)
    if ( sequence_number = receipt_sequence_number ) then
    begin
        send Ack block to indicate SUCCESS;
        advance receipt_sequence_number;
        return SUCCESS;
    end (* fi *)
    else
        send Nak block with SEQUENCE_ERROR;
end;
```

Figure 2.3

2.2 Design Consideration

Since our application required data flow from the IOP-end to the PC-end only, we decided not to develop the request facility at the IOP-end. For the same reason no request

handler is supported at the PC-end. This was because such an implementation will increase the code and more STATE related checks [cf. section 2.3] will have to be done. This could slow down the whole process of transaction. Because we require this library in a real time environment, we could not afford it. But while implementing it we have taken enough care to enable easy enhancement of the implemented protocol, to provide request from IOP end also.

There is another aspect of the protocol that we have not implemented yet, i.e., source and destination, since both are fixed at present. This aspect needs to be provided when multiple senders and receivers are supported. The reason for this is precisely the same as above. And also that there is no concept of source and destination at the PC-end. But this can be implemented with ease at the IOP-end [cf. Section 5.1]. The actual structure used by the protocol for transmitting data packets is given below:

```
typedef struct {
      unsigned char      cntl_byte;
      unsigned char      seq_num;
      unsigned char      data_size;
      char               info [ MAX_DATA_LENGTH ];
} Packet;
```

The first three fields of the structure stand for control byte, sequence number, and data length respectively. The reader may wonder where the other data attributes and the command request attributes are defined. These are maintained in a structure within the packet which protocol transmits. This structure is called Creqres [cf. Appendix H], acronym for Command REQuest and RESponse. Note that this data structure is just like any other data for the lower layer of the protocol. This data structure is interpreted by the middle layer. Also note that there are only two layers of the protocol at the IOP-end. The top layer does not exist as there is no user interface at this end. This structure is given below:

```
typedef struct {
      unsigned char      msg_length;
      unsigned char      msg_type;
      unsigned char      command;
      unsigned char      seq_num;
      long               status;
} Creqres;
```

This structure plus the actual data is placed in the array defined in the structure Packet.

The data length in the two structures represents two different lengths. In the former structure it represents the total length, while in the latter it represents the length of the actual data. Similarly, the sequence numbers in the above structures carry different meaning. The sequence number in the structure Packet represents the successfully transmitted sequence number. But the sequence number in the structure Creqres is used when the data packets exceed the maximum length supported by the protocol and they are required to be transmitted in quantums. Creqres is the structure with which the receiver can identify whether the packet has a request or a response by looking at the field 'msg_type'. The field 'command' gives the identity of the request. The field 'status' in Creqres is used only while responding to a request. This gives the status of the response, i.e., it gives the type of the error that had occurred, if any, else will give success. Note that these errors are different from the ones that have been mentioned in section 2.1. They would represent wrong request, such as, bad file name etc. This structure is solely used by the middle layer of the protocol. Different types of request commands are supported by the SOAP protocol and they have been defined in the header file soapint.h (not all of the commands have been implemented). Finally, a checksum will follow. Except the control byte all the other bytes in the Packet structure are used in computing the checksum.

There is a severe overhead in the implementation that could not be avoided. The PC supports an Intel processor that stores the MSB first and LSB as the last byte. While the IOP machine which supports Motorola 68010 does just the opposite way. It requires that all the information that are not 'chars', (one byte) be swapped while transmitting to the either of the end. Such work is completely handled at the PC-end and the IOP-end is transparent to this. For this reason we have tried keeping only chars in the SOAP transaction structures. But the user is required to take care of this while he transmits or receives packets. This overhead is simply unavoidable.

In the Nak and Ack block structures we have put some null bytes (don't care bytes). It can be attributed to the fact that at the IOP-end four bytes are allocated even if the structure is of three bytes. Whereas, at the PC-end the number of bytes to be allocated is precisely equal to the size of the structure. Hence to maintain compatibility we have inserted don't care bytes. Note that both the Ack and Nak block are of same size. They have been given below:

```
typedef struct {
      unsigned char      cntl_byte;
      unsigned char      seq_num;
      unsigned char      code;
      char               null; (* NOT USED *)
} Nak;

typedef struct {
      unsigned char      cntl_byte;
      unsigned char      seq_num;
      char               null; (* NOT USED *)
      char               filler; (* NOT USED *)
} Ack;
```

The user can select any Baud rate he wants, but the default Baud rate is 9600. The other settings are one stop bit and 8 bits data. The reason for keeping 8 bits data is to enable transmission of non-ASCII data. Since the data is of 8 bits we have kept no-parity.

The IOP-end basically does disc related I/Os. Still we have eschewed using buffered call, e.g., fread. It is so because further 'freads' may not get the data that have actually got modified in the file. For similar reasons we cannot perform anticipatory data fetches. These are some of the bindings imposed by real time systems.

The reader may also note that we have kept the maximum length of the data that can be transferred as 128 bytes. Not that data beyond this cannot be transmitted but the Soap Library routines will split the data in quantums of 128 bytes for transmission. This restriction can be elaborated by the fact that the IOP-end server is basically an application layer over the UNIX driver and hence packets more than the buffer size of the driver can get corrupted, in fact, can get over-written. But as far as the transfer from the IOP-end to the PC-end is concerned this restriction can be lifted because the PC-end maintains its own buffer for transmission and reception of data.

Extreme care has to be taken while transmitting the data type 'int'. This is because at the PC-end 'int' is of two bytes and at the IOP-end it is of four bytes. Either the user should transmit 'long' instead of 'int' or he should interpret them appropriately after transmission. The protocol is transparent to such transfers. The data is only interpreted as stream of bytes

as far as the protocol is concerned.

2.3 The PC End

The lower level protocol at the PC-end is interrupt driven. Two interrupts have been developed to handle the incoming and outgoing packets. The reasons for such an implementation is well explained by the following lines. "When events occur unpredictably (or 'asynchronously' in computer jargon), there are two ways to detect them. First, the program can check periodically to ascertain if an event has occurred. If so, program acts on it in the appropriate manner, then resumes program execution. This method, known as *polling*, wastes the time of both the processor and the programmer. In the second method, the event itself notifies the program that it has occurred and the program performs the required service when it sees fit. This is the *interrupt* method".[1]

"Even though the ROM BIOS, standard on all the MS-DOS systems, and MS-DOS itself include some support for programming the RS- 232C ports (for example, interrupt number 14h) this support, ... , is not adequate for high speed communications".[2]In fact, these interrupts fail to work properly even at 1200 Baud rate, i.e., overrun may occur. Since both MS-DOS and ROM BIOS provide no useful facility for serial communication, we developed our own interrupt handlers to handle the serial communication from COM1 and COM2 (note COM3 and COM4 can't be made interrupt driven) by directly handling the 8250 Universal Asynchronous Receiver Transmitter (UART) and the 8259 Peripheral Interrupt Controller (PIC). The reader is referred to Appendix C and Appendix D for notes on UART and PIC respectively.

Both the interrupts maintain separate ring buffers on which they operate. The interrupts have been actually written in 'C' as functions. These functions are called from routines written in assembly which perform the interrupt handler entry and exit operations.

There exists three buffers each for both, receive and transmit, separately. They

---

1. [5] p.314.
2. [1] p.453.

correspond to the three layers of the SOAP. The buffer corresponding to the top layer is the buffer into which data will be finally placed or picked up for transmission. The buffers corresponding to the lower layer are the ring buffers which the interrupt handlers use. The buffers corresponding to the middle layer are intermediate buffers. When the data is copied from the ring buffer to the buffers corresponding to the middle layer, or vice versa, the interrupts are temporarily disabled so that the ring buffers don't get modified while copying is taking place. The need for copying from the ring buffer to the middle layer buffer is that the read pointer to the ring buffers could get modified and moreover the bytes of the packet may not be contiguous in the ring buffer (since no buffer can be physically circular).

There is another feature with the SOAP protocol that has not been discussed in section 2.2. The synchronization between the sender and receiver is also achieved by STATE checks. There exists three types of states: STATE_SEND, STATE_RECV, and STATE_IDLE. From the STATE_IDLE it can switch over to either of the other two state and only Data packets will be accepted in this state. In STATE_SEND all Data packets are ignored while in STATE_RECV all Ack/Nak are ignored. Appropriate messages are also sent so that synchronization takes place.

If the ring buffer overflows the Data packets get ignored while the previous packets are ignored in case of valid Ack/Naks. Also note that both the Data packets and the control packets are placed in the ring buffer. The advantage of doing so is that automatic sequencing is maintained and furthermore it guarantees that packets received from IOP-end are according to the request put forward.

2.4 The IOP End

This part of the protocol has been written as an application layer over the UNIX driver. The idea of implementing this part as an application layer was that the IOP-end required only to serve the requests generating from the PC-end. Hence we have put minimum effort and at the same time have tried to develop a good request handler. This part of the protocol only waits for the request commands from the PC-end and issues no request to the PC- end. The process here is designed such that it will start polling the specified terminal port. The PC's COM port can be directly attached to this terminal port to enable the two machines to

communicate.

Here we have developed two processes to achieve the task in hand. The two processes are called 'pcserver' and 'pcreader'. The pcserver 'forks' the pcreader when it is initiated and they then use a 'pipe' already established by 'pcserver' for inter-process communication. Infact both these processes are 'exec-ed' by another process called 'main'. Main first 'forks' and 'execs' pcreader and then 'execs' pcserver. It has been implemented in this fashion because in UNIX the 'exec-ed' process is overlayed on the process 'exec-ing' it. Hence for obvious reasons we wanted the 'exec-ing' process to be small in size. The job of the pcreader is to poll the terminal port to receive the packets coming from the PC-end and validate them before sending them to the pcserver through the 'pipe' established. The pcserver picks the packets from the 'pipe' servers the request if it is a request command. After having sent the requested packet it waits in an infinite loop on the pipe for Ack/Naks. It is the job of the pcreader to Ack/Naks all the Data packets that come in from the PC-end.

The pcserver sets a 'signal' to catch the death of the child (pcreader) and whenever he gets this 'signal' he commits suicide, i.e., he exits. The transmission sequence number is maintained by the pcserver process and while the receipt sequence number is maintained by pcreader. The re-sequencing requests are handled in the following manner. Whenever the PC-end requests for a re-sequencing the pcreader resets his receipt sequence number and also signals the pcserver to reset his transmission sequence number.

For the request commands the pcserver calls a routine that identifies the request and serves it. We may also note here that the GIM would require information from two types of file. One being the UNIX stream files and the other being C-ISAM files [cf. Appendix B]. The UNIX stream files and the C-ISAM files are dealt separately. For the UNIX stream file it keeps the last five files opened for which the request came. While in the case of the C-ISAM file it keeps only the last three files opened. Such an implementation would help in promptly serving the PC requests. Figure 2.4 through Figure 2.15 give the implementation of this side of the protocol in pseudo code. All the UNIX system calls have been put in "''" - for example, close call is represented as 'close'. The reader is referred to Appendix F, or [26]-[29] for details on this system calls.

```
program main; (*main routine*)
begin
      set 'signal' to ignore SIGINT;
      open_device(); (* Figure 2.5 *)
      init_device(); (* Figure 2.6 *)
      create pipe through 'pipe' system call;
            'fork' and 'execlp' READER_PROCESS and
            pass pipe_write_file_descriptor as
            command line argument;
      'close' pipe_write_file_descriptor;
      'execlp' the SERVER_PROCESS and pass the
            child_process_identity as command line
            argument;
end.
```

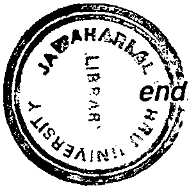


Figure 2.4

```
procedure open_device; (* open the terminal device *)
begin
      'open' terminal device;
      connect the the device_file_descriptor to stdout;
end;
```

Figure 2.5

TH-2886

```
procedure init_device; (* initialize the terminal settings *)
begin
      call 'ioctl' to get present terminal settings;
      save this settings;
      make changes for the required terminal settings
            in the above obtained information;
      call 'ioctl' with this new information to change
            terminal settings;
end;
```

Figure 2.6

Dissertation
681·3·06:621·3·06
V82
Re

```
program reader; (* READER_PROCESS *)
begin
      validate the pipe_write_file_descriptor;
      call 'plock' to lock into memory;
      call 'nice' to increase priority;
      for ( forever ) do
      begin
            read one character from port;
            case ( above_character )
            begin
            SOH:
```

```
                    read port for remaining packet;
                    call process_pkt()  (* Figure 2.7 *)
                            to validate the packet;
                    if ( valid_packet ) then
                            place it in the pipe;
                    continue;
            ACK:
            NAK:
                    read port for the remaining block;
                    place it in the pipe;
        end;
end.
```

Figure 2.7

```
procedure process_pkt;  (* validate the packet and respond *)
begin
    compute check sum;
    if ( computed_check_sum <> packet_check_sum ) then
    begin
            send Nak with CHECK_SUM_ERROR;
            return FAILURE;
    end;
    if ( packet_sequence_number <> receipt_sequence_number ) then
    begin
            if ( packet_sequence_number <> 0 ) then
            begin
                    send Nak with SEQUENCE_ERROR;
                    return FAILURE;
            end;
            reset receipt_sequence_number; (* request for reset *)
            flush the pipe of all bytes;
            call 'kill' to signal parent to reset transmit_sequence_number;
    end;
    send Ack;
    increment receipt_sequence_number;
    return SUCCESS;
end;
```

Figure 2.9

```
program server; (* SERVER_PROCESS *)
begin
    set 'signal' to catch death_of_child;  (* i.e. READER_PROCESS *)
    on catching this signal call mourn_chlds_death(); (* Figure 2.11 *)
    process_messages(); (* Figure 2.12 *)
```

```
end.
```

Figure 2.10

```
procedure mourn_chlds_death;
begin
      print message;
      'exit';
end;
```

Figure 2.11

```
procedure process_messages;
begin
      if ( 'setjmp' returns 0 ) then
      reset_tx_seq(); (* Figure 2.13 *)
      for ( forever ) do
      begin
            read from pipe sizeof(Ack / Nak block) bytes;
            if ( control_byte = SOH ) then
            begin
                  read from pipe the remaining bytes;
                  cmd_interpret(); (* Figure 2.14 *)
                  continue;
            end;
            read pipe for one byte;
      end;
end;
```

Figure 2.12

```
procedure reset_tx_seq; (* reset transmit_sequence_number *)
begin
      set 'signal' to catch 'kill' signal from child_process;
      reset 'alarm';
      reset transmit_sequence_number;
      'longjmp';
end;
```

Figure 2.13

```
procedure cmd_interpret; (* serve the request command *)
begin
      identify the request;
      call the appropriate function to get required
            information;
      (* the function below has been logically placed here *)
      (* in fact it is called by the function called above ! *)
```

```
        snd_data(); (* Figure 2.15 *)
end;
```

Figure 2.14

```
procedure snd_data; (* transmit the data packet *)
begin
    form the packet with proper header
    and check sum;
    flush the pipe from unwanted bytes;
    repeat
        write the data packet to the port;
        set 'signal' to catch time_out alarm;
        set 'alarm' to TIME_OUT seconds;
        reset time_out_flag;
        for ( forever ) do
        begin
        if ( time_out_flag = TRUE )
            break;
        if ( read pipe fails ) then
        begin
            if ( failure due to 'signal' ) then
                continue;
            fatal;
        end;
        case ( control_byte )
        begin
        SOH:
            read the remaining packet;
            continue; (* don't bother about above packet *)
        ACK:
            if ( ack_sequence_number = transmit_sequence_number ) then
            begin
                reset 'alarm';
                increment transmit_sequence_number;
                return SUCCESS;
            end;
            continue;
        NAK:
            reset 'alarm';
            if ( INVALID_STATE_ERROR ) then
                return FAILURE;
            if ( SEQUENCE_ERROR ) then
            begin
                (* lost Em syndrome *)
                if ( transmit_sequence_number + 1 = nak_sequence_number ) then
                begin
                    increment transmit_sequence_number;
```

```
                    return SUCCESS;
                end;
                continue;
            end;
        end;
    end;
until ( retry_count > maximum_retry_count );
reset 'alarm';
return FAILURE;
end;
```

Figure 2.15

2.5 SOAP Library routines

This section of the chapter explains the routines available with the SOAP Library. Only some of the major routines have been listed in the figures 2.16 through 2.20 in pseudo code to give an an overall view of the library. In what follows, we have taken one of the SOAP's standard routine, i.e. soap_fetch(), to explain how all the three layers of the protocol combine to achieve data transfers. The other SOAP Library routines have been implemented in similar fashion. The status of the transmit and receipt ring buffers are kept in the following data structures:

```
/*
    Data structure to save the received characters by reading the COMM port.
*/
typedef struct Rxbuf {
    short           count; /* Number of characters available */
    unsigned char   p_avail; /* Number of packets available */
    unsigned char   p_sts; /* Flag to indicate that last recv packet is to be ignored */
    unsigned char   p_typ; /* Type of the pkt: ACK,NAK or DATA */
    unsigned char   *wptr; /* Pointer to the next free location */
    unsigned char   *rptr; /* Pointer to location from where next character must be read. */
    unsigned char   *p_beg; /* Pointer to the beginning of the current packet being received */
    unsigned short  p_len; /* Length of the current packet */
    unsigned short  p_size; /* Expected size of the data packet */
    unsigned char   p_csum; /* Ckecksum of the packet */
    short           rx_seq_no; /* Sequence number of the next MSG */
    unsigned char   buffer[ MAX_DPKT_SIZE * 4 ];
} Rxbuf;

/*
    Data structure to save characters to be outputted to the COMM port.
*/
```

```
typedef struct Txbuf {
    short           count; /* Number of characters available */
    short           tx_seq_no; /* transmit sequence number */
    unsigned char   *wptr; /* Pointer to the next free location */
    unsigned char   *rptr; /* Pointer to location from where next character must be read. */
    /*
        Buffer to save the characters to be written into the port.
    */
    unsigned char   buffer[ sizeof( Packet ) * 2 + sizeof( Ack ) ];
} Txbuf;
```

The following data structure is maintained to keep information about the COM ports. Note that we have separate global variables for both the COM ports.

```
typedef struct Ctlport {
    void            (far * isr )();  /* ISR routine */
    short           port_base; /* Base address of the port */
    unsigned char   state; /* State of communication */
    Rxbuf           rx; /* receipt ring buffer information */
    Txbuf           tx; /* transmit ring buffer information */
} Ctlport;
```

```
procedure soap_fetch( filepath, offset, nbytes, buffer );
begin
    fill Creqres structure in request_buffer;
    soap_send( active_port, request_buffer, message_length ); (*  Figure 2.17 *)
    if ( soap_send() fails ) then
        return FAILURE;
    for ( forever ) do
    begin
        soap_recv( active_port, response_buffer, TIME_OUT ); (* Figure 2.18 *)
        if ( key_board_hit ) then
            return FAILURE;
        if ( soap_recv() fails ) then
            return FAILURE;
        if ( Creqres sequence_number not proper ) then
            return FAILURE;
        if ( any other error ) then
            return FAILURE;
        copy data from response_buffer to user buffer;
        if ( all required bytes read ) then
            return SUCCESS;
    end;
end;
```

Figure 2.16

```
procedure soap_send( port_id, data_buffer, data_size );
begin
    copy data_buffer to local_buffer and form Packet;
    note current time;
    for ( forever ) do
    begin
        set state to STATE_SEND;
        flush receipt_ring_buffer;
        _write_port( port_id, local_buffer, packet_size ); (* Figure 2.19 *)
        if ( _write_port() fails ) then
        begin
            set state to STATE_IDLE;
            return FAILURE;
        end;
        for ( forever ) do
        begin
            if ( TIME_OUT ) then
            begin
                if ( retry_count > 0 ) then
                begin
                    decrement retry_count;
                    break;
                end;
                set state to STATE_IDLE;
                return FAILURE;
            end;
            _read_port( port_id, packet ); (* Figure 2.20 *)
            if ( _read_port() fails ) then
                continue;
            case ( control_byte )
            begin
            ACK:
                if ( packet_sequence_number = transmit_sequence_number ) then
                begin
                    increment transmit_sequence_number;
                    return SUCCESS;
                end;
            SOH:
                continue;
            NAK:
                decrement retry_count;
                if ( retry_count  0 ) then
                    return FAILURE;
            end;
            break;
        end;
    end;
end;
```

Figure 2.17

```
procedure soap_recv( port_id, buffer, TIME_OUT );
begin
    note current time;
    repeat
        _read_port( port_id, local_buffer ); (* Figure 2.20 *)
        if ( _read_port() fails ) then
            continue;
        if ( control_byte = SOH ) then
        begin
            copy local_buffer to buffer;
            return SUCCESS;
        end;
    until ( TIME_OUT );
end;
```

Figure 2.18

```
procedure _write_port( port_id, buffer, packet_size );
begin
    if ( no space in transmit_ring_buffer ) then
        return FAILURE;
    disable interrupts;
    copy buffer to transmit_ring_buffer;
    update transmit_ring_buffer write_pointer;
    if ( no characters available in transmit_ring_buffer ) then
        enable transmit_interrupt;
    update number of characters available in transmit_ring_buffer;
    enable interrupts;
    return SUCCESS;
end;
```

Figure 2.19

```
procedure _read_port( port_id, packet );
begin
    if ( no packets available ) then
        return FAILURE;
    save current receipt_ring_buffer read_pointer;
    copy data from receipt_ring_buffer to packet;
    disable interrupts;
    if ( receipt_ring_buffer read_pointer not changed ) then
    begin
        update receipt_ring_buffer read_pointer;
        enable interrupts;
        return SUCCESS;
```

```
        end;
        enable interrupts;
end;
```

Figure 2.20

Chapter 3

The CGRAPHIC Library

This chapter explains the CGRAPHIC Library routines, their algorithm and implementation. It might be useful to go through Appendix E before reading this chapter. This Appendix explains about the EGA card. The basic idea of developing the graphic library was the need to have specialized routines serving our specific requirements. Our requirements were to have routines that had various styles and types for filling regions, strong string manipulating functions, Cursor control functions, a character set in a 8 x 8 matrix, to store and restore graphic screens, etc. Since all of these routines were required to be fast, as they would be used in a real time environment, it required to handle the EGA directly instead of using the MS-DOS interrupts (for the video, int 10h) as they are very slow. "On an PC/AT, the EGA BIOS will put 2.65 dots on the display in 1 millisecond (2.65 dots/ms)".[1] And whereas one can gain speed of over 200 percent above the MS-DOS interrupts. So we thought of developing a complete library having even the standard available routines. This approach would make available a complete library for other application programmers. We have built all the routines as fast as they could be made to be. This was achieved by directly handling the EGA card. The library is tailored to IBM EGA controlling a high resolution monitor. But these routines are fully independent of the operating system hence could be

---

1. [1] p.365

ported onto some other environment without significant changes. In fact, they can be ported onto systems having a different hardware. This is because, all the hardware related functions have been used as MACROs hence it would require only to change these MACROs without having to do major changes in the actual implementation. The library routines are available with 'C' interface only, i.e., they can be called directly from any 'C' program.

3.1 Library structure

The functions available with the CGRAPHIC Library can be classified into two categories: lower level routines and macro routines. The lower level routines are the basic building blocks of the library while the macro routines are standalone, i.e., they are complete in themselves and independent of each other. Hence macro routines can be called in any sequence. However, they sometimes lead to redundant operations. The lower level routines are to be used carefully in a definite order. But they will eliminate the redundant operations and thereby speeding up the application. As mentioned earlier the lower level routines start with an 'underscore' and directly manipulate the EGA registers and access the EGA display memory.

The application programmer can enhance on the existing library by using these lower level routines and in turn can maintain reasonable speed of his application program. To explain this feature consider the following simple example. Suppose the user wants to draw a hollow cube (not supported by the library) he has to use the following routines.

```
{
    boxshell ( co-ords. of 1st rectangle, color );
    boxshell ( co-ords. of 2nd rectangle, color );
    line ( co-ords. of 1st edge, color );
    line ( co-ords. of 2nd edge, color );
    line ( co-ords. of 3rd edge, color );
    line ( co-ords. of 4th edge, color );
}
```

Instead, he can use some of the lower level and core routines to achieve this. Note that the color is being

```
{
    _initega();
    _setcolor( color );
    _boxshell ( co-ords. of 1st rectangle );
```

```
        _boxshell ( co-ords. of 2nd rectangle );
        _line ( co-ords. of 1st edge );
        _line ( co-ords. of 2nd edge );
        _line ( co-ords. of 3rd edge );
        _line ( co-ords. of 4th edge );
        _resetega();
}
```

passed to the library routines six times in the former program segment. Moreover, the library will set and reset the EGA, and select and drop the color with each call. The latter program segment demonstrates how this redundant operations can be avoided. But it should be noted that no CGRAPHIC macro routines or MS-DOS calls involving the EGA be called between calling the core routines _initega() and _resetega(). Now the user can save this cube drawing function in the library and its speed will be comparable to other routines in the library.

There is another feature with this library. The CGRAPHIC Library comes along with another library DGRAPHIC. The two libraries contain the same routines but the difference with them is that the DGRAPHIC Library routines make checks for valid parameters and other required checks. So DGRAPHIC Library can be used while program development. The user can switch over to the CGRAPHIC Library when the application program is to be made operational. This aspect of the library has been discussed in greater detail in the last section of this chapter. The macro routines always return (0) for success and (-1) for failure (most often parameter error). The parameter validation is however optional and once an application has been debugged, it may be skipped. The lower level and core routines are void (does not return anything) and never does any parameter validation.

The CGRAPHIC Library normally supports graphic mode 16 of the IBM EGA (High resolution: 640 x 350) but it can be configured to the graphic mode 14 of the IBM EGA (Medium resolution: 640 x 200) with a re-compilation of the library routines. The CGRAPHIC Library supports device co-ordinates (i.e., physical co-ordinates, one unit of length representing one pixel - picture element). Moreover, the horizontal direction (left to right) is treated as positive X-axis and the vertical direction (top to bottom) as positive Y- axis, with the origin of the co-ordinate system at the top-leftmost pixel.

Since the library is available with 'C' interface the user will be required to include some Header files while using the library routines. The following Header files are supplied to the

user (the other Header files in the Appendix H are internal to the library).

<graph.h> This Header file declares all the CGRAPHIC functions with their parameters. It also defines all the parameter mnemonics.

<screen.h>
<screen14.h>
<screen16.h> These files define the screen size attributes in IBM EGA graphic mode 14 and 16 respectively.

<debug.h> This Header file is required only when the user wishes to write his own error handling routines. Also refer to last section of this chapter for more details.

3.2 Core Routines

This section explains some of the core routines that allow to configure the IBM EGA so that the CGRAPHIC lower level routines can be called. They directly access the EGA registers to do their part of the job. These core routines are used through out the library. Though they are the basic routines, they still don't remain to be independent. Before any of these routines are called the EGA has to be set to High Resolution graphic mode 16 or the Medium Resolution graphic mode 14 [cf. section 3.7]. The user is also required to take care of the order in which he calls these routines. The routines are explained below. Their usage and a brief note on them is given. All of the functions listed in this section access the graphic controller register of the IBM EGA.

Function: _initega - initialize the EGA
Synopsis: # include <graph.h>
void _initega();
Notes: This function initializes the EGA so that the lower level routines of the CGRAPHIC might work. It should be called prior to calling any lower level or core routine. It actually initializes the EGA to the CGRAPHIC settings.

Function: _resetega() - reset EGA
Synopsis: # include <graph.h>

void _resetega();

Notes: This function resets an initialized EGA so that the settings used by CGRAPHIC Library routines may not clash with MS-DOS functions. It is to be called after calling a set of lower level and core routines.

Function: _setcolor - set a color and a mix

Synopsis:
```
# include <graph.h>
void _setcolor ( color, mix );
int color;
int mix;
```

Notes: This function sets a specified color and mix option for the subsequent lower level routines (where applicable). It assumes that the EGA has been already initialized. The IBM EGA allows 16 color out of the 64 colors. The mix operation means the operation required with this color against the existing background color. There are four kinds of mix operations - Replace, Xor, Or, and And. See Appendix E for details. The mnemonic for color and mix are defined in the header file <graph.h>.

Function: _setwriteop - set write operation.

Synopsis:
```
# include <graph.h>
void _setwriteop( writeop );
int writeop;
```

Notes: The IBM EGA supports three kinds of writing operation on the EGA display memory. The default write operation both for MS-DOS and CGRAPHIC functions is (0). The write operation (1) is used for filling and write operation (2) is for copying one area of the EGA display memory onto another area. Look up [1], [15], and [20] for details and also Appendix E.

3.3 Geometric Routines

This section explains some of the geometric functions available with CGRAPHIC Library. We have taken utmost care to keep this routines as fast as possible. Most of the routines listed in this section use one property of the IBM EGA quiet frequently - each byte in the EGA display memory represent eight consecutive horizontal pixels on the screen. The

is required look up Appendix E for the configuration of the EGA display memory and also [1], [15], and [20]. All the higher level routines listed in this section have corresponding lower level routine that directly access the graphic controller register and the display memory of the IBM EGA. The higher level routines initialize the EGA, sets color and mix, and do some other routine job. The lower routines assume that the EGA has been initialized, and color and mix has been set. The higher level routines also do debugging jobs, which is however optional. The actual implementation of the algorithm corresponding to a higher level routine is basically carried out by the corresponding lower level routine. The idea of implementing the library in this fashion was to make available the lower level routines to the user to enable him to enhance the library to serve his requirements. Some of the CGRAPHIC routines are listed below. They describe the usage and the implementation. We have tried to club up the notes related to a group routines after their usage have been specified.

Function: point and _point - plot a point

Synopsis:

```
# include <graph.h>
int point ( x, y, color, mix );
void _point ( x, y ); int x, y;
int color;
int mix;
```

Function: verline and _verline - draw a vertical line

Synopsis:

```
# include <graph.h>
int verline ( x, y1, y2, color, mix );
void _verline ( x, y1, y2 );
int x;
int y1, y2;
int color;
int mix;
```

Function: horline and _horline - draw a horizontal line

Synopsis:

```
# include <graph.h>
int horline ( y, x1, x2, color, mix );
void _horline ( y, x1, x2 );
```

```
int y;
int x1, x2;
int color;
int mix;
```

Notes: The function point plots a point at the co-ordinate (x, y). In the functions verline() and horline() the first parameter represents the distance from the Y-axis and X-axis respectively. The next two parameter represent the length of the line.

Function: line and _line - draw a line of any slope

Synopsis:

```
# include <graph.h>
int line ( x1, y1, x2, y2, color, mix );
void _line ( x1, y1, x2, y2 );
int x1, y1;
int x2, y2;
int color;
int mix;
```

Function: sline and _sline - draw a stylish line

Synopsis:

```
# include <graph.h>
int sline( x1, y1, x2, y2, pts, blk, color, mix);
void _sline ( x1, y1, x2, y2, pts, blk );
int eline( x1, y1, x2, y2, mask, color, mix );
void _eline( x1, y1, x2, y2, mask );
int x1, y1;
int x2, y2;
int pts, blk;
unsigned long mask;
int color;
int mix;
```

Notes: The function line() uses the Bersenham's Algorithm to draw a line from (x1, y1) to (x2, y2). Refer to [16] and [30]. The function sline() draws a broken line joining the points (x1, y1) and (x2, y2). The parameters pts and blk represent the number of points to be light together and the number of consecutive blanks for

the broken line. This function also uses the Bersenham's Algorithm. Both the function given above can draw line of any slope. The function eline() is again a line drawing function. But this work on a mask given for the line style. That is, the mask 0xaaaa will produce a dotted line.

Function: boxshell and _boxshell - shell of a box

Synopsis:

```
# include <graph.h>
int boxshell ( x1, y1, x2, y2, color, mix );
void _boxshell ( x1, y1, x2, y2 );
int x1, y1;
int x2, y2;
int color;
int mix;
```

Notes: This function draws shell of a box (rectangle). The parameters (x1, y1) and (x2, y2) represent the diagonally opposite co-ordinates of the box. The function takes help of the functions _verline() and _horline().

Function: _band and _multiband - paint a vertical band

Synopsis:

```
# include <graph.h>
void _band ( xoff, mask, sy, ly );
void _multiband( xoff, nmasks, pmasks, sy, ly );
int xoff;
int nmasks;
unsigned char mask, *pmasks;
int sy, ly;
```

Notes: These are highly specialized routines. The function _band() draws a band of eight pixels (corresponding to a single byte in the EGA display memory) in width and with top and bottom given by sy and ly respectively. The mask represents the bit mask to be used to draw the band. The parameter xoff gives the distance of the band in bytes (remember eight consecutive horizontal pixels represent one byte in EGA display memory) from the Y-axis. The function _multiband() is also similar to _band(), it operates on a array of bit masks by using the same bit masks after every nmasks lines in the band. The parameters nmasks and pmasks represent the number of masks and the pointer to the array of masks

respectively.

Function: boxfill and _boxfill - filling a box

Synopsis:

```
# include <graph.h>
int boxfill ( x1, y1, x2, y2, color, mix );
void _boxfill ( x1, y1, x2, y2 );
int x1, y1;
int x2, y2;
int color;
int mix;
```

Function: connect - connect a sequence of points

Synopsis:

```
int connect ( lmask, npts, ppts, color, mix );
void _connect ( lmask, npts, ppts );
unsigned long lmask;
int npts;
int *ppts;
int color;
int mix;
```

Notes: This routine connects a sequence of 'npts' points by straight lines of the style given by the line mask parameter lmask. This routine can be used to draw polygons etc.

Function: polyfill - fill a polygon

Synopsis:

```
int polyfill ( npts, ppts, color, mix );
int npts;
int *ppts;
int color;
int mix;
```

Notes: This function fills a box by the specified color and mix. Here the user is given more options then just filling the box with specified color and mix. This function supports different types of filling a box. The type of filling required can be chosen by ORing the mnemonic (See dithering options in header file <graph.h>) with the mix option. Such dithering options have been made

available by using _multiband(). The function polyfill() fills any bounded polygon described by the 'npts' points. This uses the Edge-List algorithm [cf. [30]] to fill such regions. All the dithering and hatching schemes available with boxfill are also available here.

3.4 String Manipulation Routines

Some of the string manipulating routines are listed here. These routines work on two character sets. One is CGRAPHIC character set and the other is the standard MS-DOS character set. The CGRAPHIC character set uses a dot matrix character font of dimension 8 x 8 (including separators at bottom and right edges). The MS-DOS character can be placed on one of the 80 x 25 (both in graphic mode 16 and 14) matrix on the screen and they are always placed on a black background. Unlike the MS-DOS characters the CGRAPHIC characters can be placed any where on the screen, i.e., they need not be placed at any special boundary. All ASCII character are supported and some special character are also there. Note that this character font is fixed and can't be expanded or contracted. The routines that manipulate the CGRAPHIC character set are prefixed 'print', while those manipulating MS-DOS character set are prefixed 'write'. Here too, some of the routines have a corresponding lower level routine. Some of the string manipulating routines are listed below. Their usage and a brief description as to how these routines work is given.

Function: print and _print - print a CGRAPHIC character

Synopsis:

```
# include <graph.h>
int printchar ( x, y, color, mix, ch );
void _printchar ( x, y, ch );
int x, y;
int color;
int mix;
char ch;
```

Notes: This function will print the CGRAPHIC character ch with its top-left corner at the position (x, y) by the specified color and mix. Note that there is no restriction on the position of the character.

Function: printcol and _printcol - print in a column

Synopsis:

```
# include <graph.h>
int printcol ( x, y, color, mix, string );
void _printcol ( x, y, string );
int x, y;
int color;
int mix;
char *string;
```

Function: printrow and _printrow - print in a row

Synopsis:

```
# include <graph.h>
int printrow ( x, y, color, mix, string );
void _printrow ( x, y, string );
int x, y;
int color;
int mix;
char *string;
```

Notes: These functions print a CGRAPHIC character string pointed to by the parameter string with the top-left corner of the first character placed at the co-ordinate (x, y) by the specified color and mix. The functions printcol() and printrow() print the given string in a column and row respectively. Note that the string should be null terminated.

Function: pcolfmt - print in a column (formatted)

Synopsis:

```
# include <graph.h>
int pcolfmt(x, y, color, string, style, fcolor);
int x, y;
int color;
char *string;
int style;
int fcolor;
```

Function: prowfmt - print in a row (formatted)

Synopsis:

```
# include <graph.h>
int prowfmt(x, y, color, string, style, fcolor);
```

```
int x, y;
int color;
char *string;
int style;
int fcolor;
```

Notes: These functions pcolfmt() and prowfmt() prints a CGRAPHIC character string pointed to by the parameter string in the specified color in column and row respectively. The co- ordinates x and y represent the center of the column and row in the functions pcolfmt() and prowfmt() respectively. There are several styles supported by the CGRAPHIC. These mnemonic are defined in the header file <graph.h>. Some of the style are top justify, bottom justify, and center justify at y for the function pcolfmt(). Similarly, left justify, right justify, and center justify at x for the function prowfmt(). The parameter fcolor represents the color of the the box enclosing the string. This parameter is don't care if framing by a box is not opted for. Also note that the string should be null terminated.

Function: writechar - write a MS-DOS character

Synopsis:

```
# include <graph.h>
int writechar ( x, y, color, mix, ch ); int x, y;
int color;
int mix;
char ch;
```

Notes: This function print a MS-DOS character of the given color and mix. Note that, x and y are truncated to the nearest allowed character position.

Function: writerow - write in a row

Synopsis:

```
# include <graph.h>
int writerow ( x, y, color, mix, string );
int x, y;
int color;
int mix;
char *string;
```

Function: writecol - write in a column

Synopsis:

```
# include <graph.h>
int writecol ( x, y, color, mix, string );
int x, y;
int color;
int mix;
char *string;
```

Notes: These functions writerow() writecol() prints a string pointed to by the parameter string in a row and column respectively of the specified color and mix. Note that, x and y are truncated to nearest allowed character position and they represent the co-ordinates of the first character in the string. The string should be null terminated.

Function: getstr - get a string from keyboard

Synopsis:

```
# include <graph.h>
int getstr(x, y, prompt, color, buffer, nchar );
int x, y;
char *prompt;
int color;
char *buffer;
int nchar;
```

Notes: This function prints the prompt string and reads a string with echo and with limited editing facility. The prompt and the echo string are displayed in a horizontal row starting at (x, y). The string is terminated by a Carriage Return on the keyboard. Then the string is available in the user buffer pointed to by the parameter buffer. DEL key can be used the character typed in.

3.5 Cursor Control Routines

The CGRAPHIC Library supports different types of cursors. This section explains about the routines that control the cursors. Presently the CGRAPHIC supports five types of cursors. They have been listed in the header file <graph.h>. At any given time the user can have any five cursors (with repetition) on the screen and can move any of the cursors by using the routines listed below. They give the usage and a brief description as to how they

work.

Function: cursset - set a cursor

Synopsis:
```
# include <graph.h>
int cursset( cursnum, x, y, style, color );
int cursnum;
int x, y;
int style;
int color;
```

Notes: This function set a cursor of the given style and color. It is made to appear at the location (x, y). The parameter cursnum represents the identity of the cursor that the user would like have.

Function: cursmove - move cursor

Synopsis:
```
# include <graph.h>
int cursmove ( cursnum, x, y, mode );
int cursmove;
int x, y;
int mode;
```

Notes: This function allows the user to move the cursor identified by cursnum from its present location to the specified location (x, y). The parameter mode gives the mode of movement of the cursor that the user would like to have.

Function: cursreset - reset a cursor

Synopsis:
```
# include <graph.h>
int cursreset ( cursnum );
int cursnum;
```

Notes: This function resets a set cursor identified by cursnum. The cursor is removed from the screen.

Function: _cursprint - print a cursor

Synopsis:
```
# include <graph.h>
void _cursprint ( cursnum, x, y );
```

```
int cursnum;
int x, y;
```

Notes: The user can use this function to print a cursor identified by cursnum at the location (x, y). Note that the cursor is always XORed with the background.

3.6 Mapping Routines

CGRAPHIC also supports mapping windows. Similar to the cursors the user can select a several window where ever it pleases him. And then can call the routines listed below to work on any particular window. It provides any integer type of world co-ordinates.

Function: map - defines a mapping window

Synopsis:
```
int map ( map_num, map );
int map_num;
Mapping *map;
```

Notes: This routine maps a window to the device co-ordinates. This helps in setting the world co-ordinates to any view-port. Further interaction with this mapping window can be done by referring the same mapping number 'map_num'. The structure Map is defined in the header file <graph.h>.

Function: calibrate - calibrate the mapping window

Synopsis:
```
int calibrate ( map_num, p_cal );
int map_num;
Calibration *p_cal;
```

Notes: This routine is to be called only when the map() routine given above has already been called for this map window identity. This routine basically takes care of the options involved with the mapping window such as, box around, logical axis, calibration, grid, back ground color etc. The structure Calibration is given in the header file <graph.h>.

Function: plot - plots logical points and connect

Synopsis:
```
int plot ( map_num, ln_type, pt_type, ppts, color, mix );
int map_num;
unsigned long ln_type;
```

```
int pt_type;
int *ppts;
int color;
int mix;
```

Notes: This function maps the set of points pointed by the parameter ppts to the mapping window identified by the parameter map_num. It will also connect this point by the line style given in the parameter ln_type. The parameter pt_type represents the type of point to be put where the logical point gets mapped onto the screen.

Function: bar - draw bar chart

Synopsis:
```
int bar ( map_num, bar );
int map_num;
Bar *bar;
```

Notes: This is bar chart drawing routine. It can draw several kind of bar charts such as horizontal bars, vertical bars, range, divided bar etc. The chart will be drawn in the mapping window identified by the parameter map_num. The structure Bar and other bar chart related options are given in the header file <graph.h>.

3.7 Miscellaneous Routines

This section lists some of the general routines supported the CGRAPHIC Library. Some of the routines listed here access the CRT controller register and the graphic controller register of the IBM EGA.

Function: setgmode - set graphic mode

Synopsis:
```
# include <graph.h>
int setgmode ( mode );
int mode;
```

Notes: This function sets a graphic / text mode with the IBM EGA. Note that only one graphic mode supported with CGRAPHIC Library. The user is required to set graphic mode 16/14 of the IBM EGA before using any of the routines of the CGRAPHIC Library.

Function: spread - spread a color in a window

Synopsis: # include <graph.h>

int spread(x1, y1, x2, y2, color, mix, style);
int x1, y1;
int x2, y2;
int color;
int mix;
int style;

Notes: This function spreads a color in the window specified by the diagonally opposite co-ordinates by the style specified. The filling styles are defined in the header file <graph.h>. For example, it has a option to fill a given region by putting dots randomly by using a linear congruential generator achieving a full cycle. See [22].

Function: setpage - set a page for drawing

Synopsis: # include <graph.h>

int setpage(pageno);
int pageno;

Notes: This function sets a drawing page in the display buffer of the IBM EGA. All subsequent CGRAPHIC functions will draw in that page. This page may be different from the one currently being displayed. The number of pages available depends on the memory available with the EGA card. The page number are enumerated (0) onwards. By default, page (0) is set for drawing.

Function: dispage - display a page

Synopsis: # include <graph.h>

int dispage (pageno);
int pageno;

Notes: This function displays the specified page from the display memory of the IBM EGA. This page may be different from the one set for drawing.

Function: save - save a window in a file

Synopsis: # include <graph.h>

int save (pageno, x1, y1, x2, y2, pathname);

```
int pageno;
int x1, y1;
int x2, y2;
char *pathname;
```

Notes: The user can save any window specified by the diagonally opposite co-ordinate (x1, y1) and (x2, y2) in the page given by pageno. The window will be stored as a file by the name pointed to by the pathname.

Function: restore - restore a window from a file

Synopsis:

```
# include <graph.h>
int restore ( pageno, mix, pathname );
int pageno;
int mix;
char *pathname;
```

Notes: This function read the file given by the parameter pathname for the digitized information about a window to restore it to the specified page. The window being restored can be made to overlap with the existing background with a specified mix option. The file to be restored should be created through the function save().

3.8 Debugging Aids

To achieve speed, the routines in the CGRAPHIC normally does not validate the parameters passed by the application programs. With wrong parameters, e.g. co-ordinates beyond the screen limits etc., makes the library behave unpredictably and these situation are sometimes difficult to debug. To help the user in debugging, another library DGRAPHIC exists along with CGRAPHIC. The DGRAPHIC Library contains identical routines of CGRAPHIC but with parameter validations enabled for macro functions. The user can use DGRAPHIC Library while developing application programs and switch over to CGRAPHIC Library when it is to be made operational. With DGRAPHIC, the user can validate a program in either of the two following ways.

By default, when the macro function of DGRAPHIC detects an error (usually a parameter error), it print an error message at the bottom-most line of the screen indicating-

the erring routine, the invalid parameter and the its value and then halts. The user is now given two options - either to skip the erring function, or to abort the program. This printing can be disabled and enabled using the functions setgperror() and resetgperror() respectively. The function gerror can be used to print an error message.

Secondly, the user may alternatively choose to do the error processing himself. For this purpose two global variables 'gfunction' and 'gerrorno' (both of integer type) are made available to the user. This are defined by DGRAPHIC to contain a code for the last called function and error code respectively. When the macro function returns (-1), indicating an error, the user may look into these variables. The codes used with the variables 'gfunction' and 'gerrorno' are defined in the header file <debug.h>.

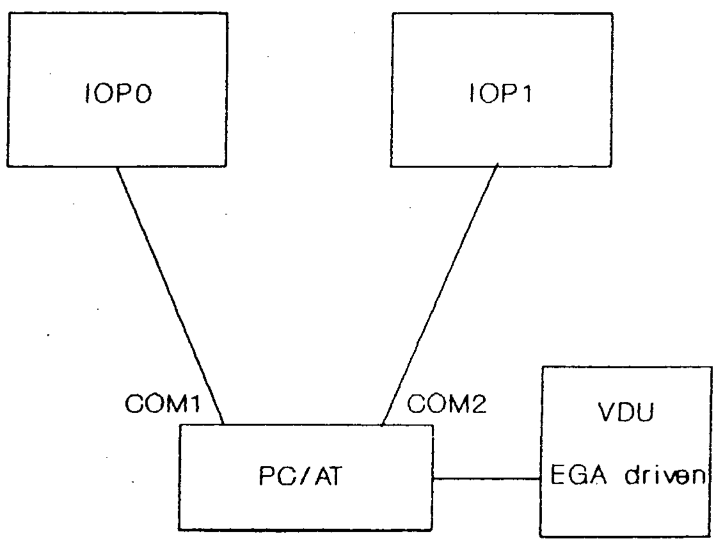
Chapter 4

The GIM

The application layer over the two libraries is called the GIM (Graphic Interface Module). This layer shows the status and performance of the MAX. Since the MAX consists of several modules and sub-modules, GIM shows the status of these modules and sub-modules in several graphic pages. The graphic display of modules and sub-modules are proportionate to their actual sizes. The statuses of the units in the modules and sub-modules are shown in different colors.

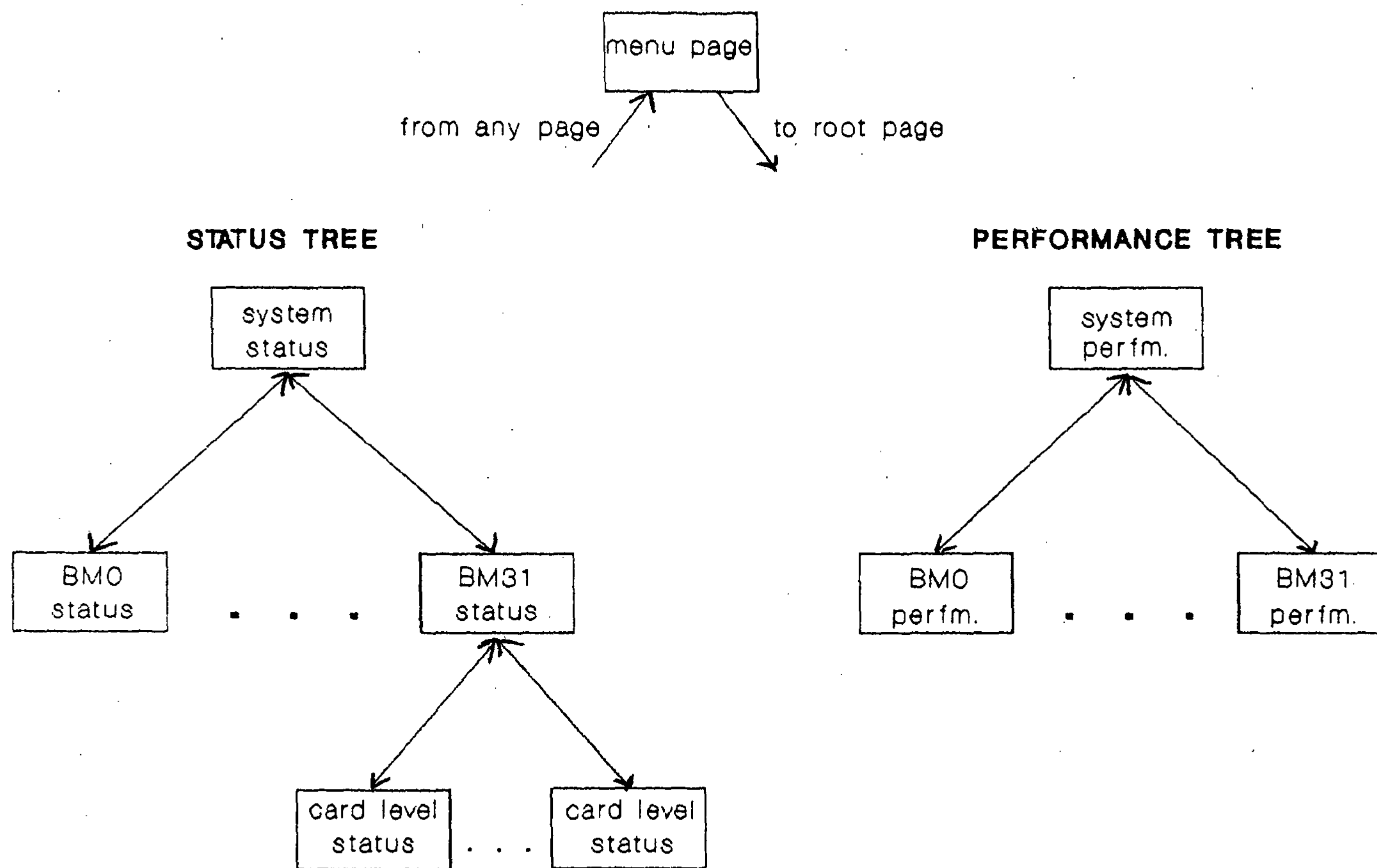
The Graphic Interface Module (GIM) uses a Enhanced Color Display (ECD) high resolution color monitor as a display device. It is connected to the C-DOT DSS through two Asynchronous Communication Interface Adapter (ACIA) links terminating on the two IOPs. See Figure 4.1. GIM collects data about the DSS through these links from either of the IOPs (whoever is 'In-Service-Active') and depicts them pictorially on the video monitor.

The GIM follows a forest structure. See Figure 4.2. That is, the status and performance of the MAX has been distributed in several trees. Each node of a tree shows the status and performance. By default, each of the sibling pages are shown in a round robin loop. The user can go to any of the sibling page, or expand on a child page, or go to the parent page. The user can also jump to the root of any of the trees in the forest through a menu page. The implementation is highly modular. This has been achieved by using appropriate data structures. Moreover, each set of different page forming routines are similar in structure. Such a modular approach also helped us in changing configuration of several pages, inserting new



GIM Hardware Configuration

Figure 4.1



GIM Software Architecture

Figure 4.2

pages, adding a sub-tree, and moreover creation of a whole tree in the forest. Not only the enhancement of GIM will be easy but maintenance will also be a simpler task.

The user is provided with several options in all the pages. He can store a page in a file or view an already stored file. He can lock onto a page, but the page will periodically (this period is tunable, presently it is tuned to 20 seconds and if the page is not locked it will display the next sibling page) get updated. There are many other such facilities provided to the user.

4.1 Hardware Requirements

The hardware requirements for the GIM package are minimal. The GIM can run on PC/XT also though the speed might slow down a little in this case. But any of the system must have a 20 MB hard disk. The other hardware requirements are an EGA card, a ECD high resolution monitor, and two ACIA ports. The EGA card and high resolution monitor are compulsory because the main objective of the package was to show the status of the system graphically, and such a package can give better performance with these supporting hardwares. The ACIA is required as the communication between the PC and IOP is required to transfer data from the IOP to show the statuses. Two ACIAs are required to maintain a fault tolerant system so that the package works unhindered.

4.2 Software Architecture

There are similar set of routines for all the pages. Some pages use the same set of routines. For example, the pages pertaining to all the Base Modules (BM) are drawn by the same set of routines but they use a global variable to know which BM is to be drawn. Each set is made up of the following routines: _getfiles(), _frame(), _putvalues(), _select() and _next(). They have been appropriately prefixed in accordance with the page they form. The main job of the _getfile() routines is to fetch the data pertaining to that particular page and place it in a buffer. The routines _frame() basically work on structures that totally describe the page settings. These structures have been built such that it is enough to modify them for changing the page setting without changing the _frame() routines. For example, the BM status page routines uses the following data structures for showing this page.An enthusiastic

reader may look up Appendix H for details.

```
typedef struct {
    char   *string; /* module name */
    int    x1, x2; /* left and right x co-ordinates */
    int    y1, y2; /* top and bottom y co-ordinates */
    int    cx, cy; /* cursor position for this object */
    char   status; /* module status (in color value) */
    char   page_no; /* page_no to expand this object */
    char   alm_id; /* index into the alm info table */
    char   alm_status; /* status of the last alarm */
} Module; /* descriptor of an module */

typedef struct {
    Stream      *stream; /* file to give status of this module */
    int         boff; /* offset for first module of this type in buffer associated with the stream */
    int         nmodules; /* no. of modules of this type */
    Module      *module; /* pointer to first module of this type */
} Modules; /* descriptor to modules of same type */
```

The routines _putvalues() pick the status related data from the buffer, filled by the corresponding _getfile() routine, and display them using appropriate colors. These routines also access the corresponding structure mentioned above. Whenever the user wants to expand on some modules being shown in that particular page the corresponding _select() routine is invoked. This routine allows the user to select only those modules that are active (since status of an inactive module cannot be shown). Each of the pages have a unique page-identity. The main routine works with the help of the routines mentioned above and invokes a set of these routines depending on the page-identity. The main routine keeps track of its where-abouts in the forest. The main routine uses the following data structure for this purpose.

```
typedef struct {
    char   *name; /* name of this display page */
    int    bg_color; /* background color */
    int    bg_style; /* style for filling color */
    int    parent_page; /* parent page no. */
    int    (far * getinfo)(); /* routine to get info. from iop */
    void   (far * drawframe)(); /* routine to draw the frame */
    void   (far * putvalues)(); /* routine to put values in the frame */
    int    (far * select)(); /* routine to select an object */
    int    (far * nextobj)(); /* routine to get the next object */
} PageFunc;
```

A good number of facility is provided to the user in the menu on each page. The options available with the menu are listed below.

- lock/unlock current page (toggle key)
- goto menu page
- goto parent page
- dump current page
- replay a dumped page
- goto next sibling page
- select an module for expansion
- stop alarm beep

The user can lock onto a page. By this the automatic transition to the sibling pages stop, and only on the demand of the user the next page is shown, which again will be locked until the user unlocks the page. Even the current date and time are shown upto minutes in every page. The user can save pages in a file or restore already saved files. He can jump to the parent page or to the main menu page where roots of all the trees in the forest are defined. And finally a quit option is also provided to quit gracefully from the package. Some of the other features of the GIM are explained in the following sections.

The GIM implementation has been explained in pseudo code in Figure 4.3 through Figure 4.13.

```
program main();
begin
    gim_setup(); (* Figure 4.4 *)
    note current time;
    while ( forever ) do
    begin
        show_page(); (* Figure 4.5 *)
        while ( forever ) do
        begin
            show current time on screen;
            if ( key board hit ) then
                if ( analyze_kb() ) then (* Figure 4.6 *)
                begin
                    change_screen := TRUE;
                    break;
                end;
            if ( TIME_SLICE over ) then
            begin
                if ( screen not locked ) then
                begin
                    if ( no parent to current page ) then (* root page ? *)
                    begin
```

```
                              set next page;
                              change_screen := TRUE;
                              break;
                         end;
                         call parent page's _next(); (* Figure 4.7 *)
                         if ( success ) then
                              change_screen := TRUE;
                    end;
                    break;
               end;
          end;
     end;
end.
```

Figure 4.3

```
procedure gim_setup();
begin
     setgmode(16); (* CGRAPHIC Library *)
     dispage(1); (* CGRAPHIC Library *)
     soap_setup(); (* SOAP Library *)
     flush alarm buffer;
     make gim screen partitions;
     display GIM version number;
end;
```

Figure 4.4

```
procedure show_page();
begin
     error_flag := FALSE;
     first_time := TRUE;
     for ( forever ) do
     begin
          check if alarms exist;
          if ( cannot communicate ) then
          begin
               call next_page's _getinfo(); (* Figure 4.8 *)
               if ( _getinfo() succeeds ) then
                    break;
          end;
          if ( error_flag = FALSE and key board hit ) then
          begin
               if ( first_time = FALSE ) then
                    return;
               flash error message;
               flush key board queue;
               continue;
```

```
                end;
                flash error message; (* cannot communicate *)
                show communication failure help_menu;
                repeat
                    if ( key board hit )
                    begin then
                        case ( character )
                        begin
                        QUIT:
                            exit GIM;
                        REPLAY:
                            replay(); (* Figure 4.11 *)
                            continue;
                        end;
                    end;
                until ( TIME_OUT );
                if ( DUPLEX mode ) then
                begin
                    cancel alarms;
                    reset next_page;
                    first_time := TRUE;
                    soap_switch(); (* SOAP Library *)
                end;
                initialize active port; (* SOAP Library *)
                change_screen := TRUE;
                error_flag = TRUE;
            end;
            if ( change_screen = TRUE ) then
            begin
                call next_page's _frame(); (* Figure 4.9 *)
                change_screen := FALSE;
                first_time := FALSE;
            end;
            call next_page's _putvalues(); (* Figure 4.10 *)
            exist_page := next_page;
        end;
```

Figure 4.5

```
procedure analyze_kb();
begin
    get character;
    case ( character )
    begin
    MENU:
        menu(); (* Figure 4.12 *)
        return SUCCESS;
    LOCK:
```

```
            toggle lock position;
            return FAILURE; (* does not mean failure ! *)
        SELECT:
            call exist_page's _select(); (* Figure 2.13 *)
            return value returned by _select();
        NEXT:
            call exist_page parent's _next(); (* Figure 4.7 *)
            return value returned by _next();
        PARENT:
            if ( no parent to exist_page ) then
            begin
                flash error message;
                return FAILURE;
            end;
            next_page := parent_page;
            return SUCCESS;
        QUIT:
            exit GIM;
        DUMP:
            get file name;
            save(); (* CGRAPHIC Library *)
            return FAILURE; (* does not mean failure ! *)
        CAN_ALM:
            cancel alarm beep;
            return FAILURE; (* does not mean failure ! *)
        DEFAULT: (* junk ! *)
            flash error message;
            return FAILURE;
    end;
end;
```

Figure 4.6

```
procedure _next();
begin
    set next page;
    set module_number if required;
    return SUCCESS;
end;
```

Figure 4.7

```
procedure _getinfo();
begin
    get IOP filename;
    call appropriate SOAP Library routine to get information;
    if ( SOAP routine fails ) then
        return FAILURE;
```

```
        put information in buffer;
        return SUCCESS;
end;
```

Figure 4.8

```
procedure _frame();
begin
        make fresh GIM screen partitions;
        access next_page's data structure and
                draw frames for all modules in this page;
end;
```

Figure 4.9

```
procedure _putvalues();
begin
        access the information buffer and
                color all modules to show status and performance;
end;
```

Figure 4.10

```
procedure replay();
begin
        while ( forever ) do
        begin
                get file name;
                restore(); (* CGRAPHIC Library *)
                show replay_menu;
                while ( forever )
                begin
                        get character;
                        case ( character )
                        begin
                        NEXT:
                                break;
                        QUIT:
                                return;
                        end;
                end;
        end;
end;
```

Figure 4.11

```
procedure menu();
begin
    draw menu_page;
    display menu_page menu;
    display all root page names;
    while ( forever ) do
    begin
        high light currently selected item;
        while ( no key board hit ) do
            update time window;
        get character;
        case ( character )
        begin
        NEXT:
            high light next item;
            continue;
        SELECT:
            set next_page to high lighted item;
        QUIT:
            return;
        REPLAY:
            replay();
            return;
        DEFAULT: (* junk ! *)
            flash error message;
        end;
    end;
end;
```

Figure 4.12

```
procedure _select();
begin
    if ( only one module in the exist_page ) then
        return FAILURE;
    point cursor to first module in the exist_page; (* CGRAPHIC Library *)
    show select_menu;
    get character;
    case ( character )
    begin
    NEXT:
        advance cursor to next module;
        continue;
    EXPAND:
        remove cursor; (* CGRAPHIC Library *)
        set next_page;
        set module_number if required;
        return SUCCESS;
```

```
        QUIT:
              show menu;
              remove cursor;
              return FAILURE;
        end;
end;
```

Figure 4.13

4.3 Alarm Display

The GIM also shows the alarms that aries in the system. It can act effectively as an alarm display panel of the total system. The alarms are kept in a C-ISAM [cf. Appendix B] file in the IOP. It gets updated as soon as the alarms arise or get rectified. This information is fetched only in the root page of the status tree, and if alarms exist it is shown by a flag at the top of the screen and also the user's attention is attracted by a beep. He can stop this beep by a cancel option provided in the menu. At all other times it is only checked whether the file has been modified since the last access, and if so the user is informed by changing the color of the alarm flag and sounding a beep which is set again in such cases. The implementation is so since picking the alarm information frequently will slow down the package.

4.4 Error Messages

The package flashes error messages whenever such conditions arise. We have kept a special error window to flash the error messages. These messages are self descriptive and can help the user in properly handling the package. If the GIM is not able to communicate with the IOPs it flashes an error message indicating the IOP to which it is not able to communicate.

4.5 Duplex Mode

The GIM works in a duplex mode, i.e., it can communicate with both the IOPs (Master and Slave) for fetching status and performance related data. The GIM will basically communicate with one of the IOPs. But it will automatically switch over to the other IOP if the COM port, for any reason, fails with the first IOP. This switching is done only if the GIM

configuration is set to DUPLEX mode. By default the GIM runs in SIMPLEX mode. The GIM has been developed for both Single Base Module (SBM) and Multi Base Module (MBM) configuration [cf. Appendix B]. This has been achieved through the conditional compilation available with 'C'.

Chapter 5

Conclusion

This chapter takes a retrospective view of the total project. Here we discuss the possible improvements and enhancements that could be made. This aspects have been discussed in the sequence in which the chapters appear in this manuscript (with the exception of the first chapter). We briefly describe as to how the enhancements could be made especially in areas where they are immediately required. Moreover, this chapter also discusses the portions of the package where implementation could be improved.

5.1 Protocol

The protocol explained in chapter 2 can be implemented to the fullest extent. That is, developing a full fledged request facility at the IOP-end and a request handler at the *PC-end*. This can be implemented by identifying whether it is a request or a response from the Creqres structure [cf. section 2.2]. At both the ends appropriate modules can be developed to handle the respective jobs. Even the 'source' and 'destination' concept in the protocol can be implemented. At the IOP-end this would require that the pcreader identifies the response and passes the packet to another module. This module should pass on the packet to the appropriate process. This can be implemented through 'message queues' of UNIX. The processes at the IOP-end can register their requests through the pcserver. The processes that would like to communicate to the PC-end should enroll themselves with the IOP-end interface, by which they can idêntify themselves, and further interaction can be done based

on this identity. As far as the PC-end is concerned there is no concept of more than one process running at a time [cf. section 5.3]. Hence it should suffice to build only a request handler at this end. Note that this protocol then should not be used in the present status of GIM. It could incur unnecessary overheads. But if further expansion of GIM requires such facility, this might prove to be very useful [cf. section 5.3]. But such an extension can have other users to whom this can prove to be highly beneficial. This configuration can then be used for IOP to IOP communication also via the PC, or even directly, i.e., by-passing the PC. Furthermore, even PC to PC communication can be established. This might require slight changes in the implementation. For example, presently, the variables that are of the type int, short, or long are swapped while transmitting to either end [cf. section 2.2]. This will not be required in PC-PC communication. These implementations can prove to be very useful.

Other improvements that could be made in the implemented protocol includes trying to reduce the copying of the data packet into buffers at various levels at the PC-end. Presently the copying is done thrice. The reason for implementing it this way was to free the lower level buffer as soon as possible. Because during this copying it a must that the interrupts are disabled, otherwise the data packet being copied could get corrupted. Another reason being that the middle layer routines will then have to take care when the packets are not contiguous (note that no buffer is physically circular) and thereby reducing modularity. But avoiding this extra copying properly can make the protocol more efficient.

5.2 Graphics

Though the graphic library is quite generalized, we feel there remains plenty of room for enhancements. The present implementation supports only device co-ordinates. But facility to set world and normalized co-ordinates could be one major enhancement. This would remove the device dependency and also help the user in working in any co-ordinate system he feels comfortable. This can be developed as a layer over the existing library working with device co-ordinates. We can straight away transform the requirement from the world or normalized co- ordinate level to the device co-ordinate level where the actual implementation can take place.

At the existing level itself the user can be provided with many more functions such as drawing graphs, drawing different kinds of statistical charts, three dimensional images,

rotation, translation etc. Another useful package can be developed with whom the user can interactively draw his graphic figures. Furthermore, a useful add-on could be a graphic plotter/printer driver that prints the files saved by the function save().

What ever routines the library supports are quite powerful as well as fast. But we realize that the CGRAPHIC Library does not fully exploit the EGA facilities, for example, character generation etc.

5.3 Application Layer

The basic structure of the whole application layer could be made better looking, this may not actually add to the efficiency of the package. That is, we can keep all the information regarding a page in a structure. We can then form a linked structure of the above structure of the sibling pages. Each of these nodes should have a pointer to the child, as well as to its parent. Such an implementation can bring the actual tree structure in the forest. But this would require a complete re-structuring of the whole package.

Since the GIM shows the real time status of a system it is important that it is made as fast as possible. Toward this end we have a suggestion. Though difficult to implement, we can introduce multitasking. This will increase the speed of the package by a noticeable degree. The multitasking facility is supported only in PC/AT. The interrupts for this is Int 15h. It only provides very primitive tools towards this end. To introduce multitasking a whole scheduler might have to be developed. The following lines echo our thoughts. "In general, MS-DOS does not support multitasking, although MS-DOS for the IBM PC AT computers has provisions for simple multitasking. Multitasking is a very powerful technique for real time system. It simplifies system design and makes it possible to design large, complex systems. A real-time system is aimed at processing several independent events that occur at random times. The event can be asynchronous and concurrent. This means that an event can occur while one is already being processed. Multitasking can be used in such systems to simplify

software design".[1]

But lastly we feel that there is always enough room for improvements no matter what the job be . . .

1. [1] p.276.

Appendix A

Glossary

68010	— Serial number of a micro processor manufactured by Motorola Inc. of U.S.A. It is a member of 68xxx series of micro processors.
ACIA	— Asynchronous Communication Interface Adapter
ACK	— Acknowledgment
ASCII	— American Standard Code for Information Interchange.
AM	— Administrative Module: A basic module of C-DOT digital switching system.
BIT	— Binary Digit.
BYTE	— A group of BITs (usually eight).
BM	— Base Module: Primary growth unit of C-DOT Digital Switching System and one of its four basic modules.
C	— A programming language having powerful capabilities for system programming.
C-DOT	— Centre for development of Telematics: India's Telecom technology Centre.
C-ISAM	— A professional package for indexed sequential access file management system.
CDOS	— C-DOT's real time operating system: Base Processor with 68010 CPU runs under CDOS and provides uniform interface to rest of the modules.
CGRAPHIC	— The graphics library developed for GIM

COM(port) — Communication port

CPU — Central Processing Unit

CM — Central Module: A basic module of C-DOT Digital Switching System.

CP — Call Processing: A number of complex functions are performed to process a telephone call. Providing dial tones etc., searching of available path for making physical connections between the called and calling party etc., all of these fall under call processing functions.

DSS — Digital Switching System: Generally used for telephone/telex exchanges designed using digital technology.

DTMF — Dual Tone Multi Frequency : A coding in which two tones on different frequencies are used for distinguishing digits dialed by a telephone user.

EGA — Enhanced Graphic Adapter

ETX — End of Text

EM — End of Medium

GIM — Graphic Interface Module

HDLC — High Level Data Link Controller: Used in C-DOT Digital Switching System for communication link control between various modules.

I/O — Input/Output

IOCM — Input Output Configuration Manager : A process which schedules audit process.

IOP — Input Output Processor: The front end computer system for C-DOT Digital Switching System built around 68010 CPU and running under UNIX. It supports a variety of peripherals such as VDU, printer, disk derive, tape derive etc.

ISR — Interrupt Service Routine

MAX — Main Automatic Exchange

MBM — Multi Base Module: A typical configuration of C-DOT Digital Switching System in which upto 32 base modules are present. Such a configuration is under going trials at Ulsoor, Bangalore, telephone exchange.

microVAX — Micro VAX: a computer system developed by Digital Equipment Corporation U.S.A.

MS-DOS — Operating System for the IBM PC and compatibles developed by Microsoft, Inc., U.S.A.

NAK — Negative Acknowledge

OOS — Out Of Service: Condition indicating that a unit is not working properly and has been removed from service.

PIC — Peripheral Interrupt Controller

PC — Personal Computer

PP — Peripheral Processing: Actual switching i.e. making the physical connection between caller and called party, and other related functions fall under this category.

RS-232 — Standard serial hardware link

SBM — Single Base Module: a configuration of C-DOT Digital Switching System in which only one base module is present such a configuration is under going trials in Delhi Cantt exchange.

SOAP — Simplest Of All Protocols. Communication protocol Library developed for GIM.

SOH — Start of Header

STX — Start of Text

UART — Universal Asynchronous Receiver Transmitter

UNIX — An operating system becoming more and more popular due to its flexibility, simplicity, portability and adaptability.

VAX — Virtual address extension: a brand name of computer system developed by Digital Equipment Corp. U.S.A.

VDU — Visual Display Unit: A gadget on which output from a computer is displayed on a television like screen.

Appendix B

Notes on C-DOT DSS

C-DOT Digital Switching System (DSS) is highly modular in structure and hardware is duplicated, for fault tolerance, at every possible level. One module of the two, is called ACTIVE and the other STAND-BY. In the event of fault a smooth switch over is performed, from active to standby. Software in addition to modularity has other features of layered and distributed processing also. Different processors running under different environments process global data such as billing data, traffic data etc. This Appendix concentrates on Hardware and software organization of C-DOT DSS.

B.1 Hardware Architecture

C-DOT family of products has four basic hardware units.

- Base Module (BM)
- Central Module (CM)
- Administrative Module (AM)
- Input Output Processor (IOP)

A typical configuration of these units is shown in Figure A.1. Although IOP is a part of AM it is discussed separately further details are provided in C-DOT DSS ARCHITECTURE [6]-[8].

Base Module

BM is primary growth unit of C-DOT DSS. BMs may differ in the types and quantities

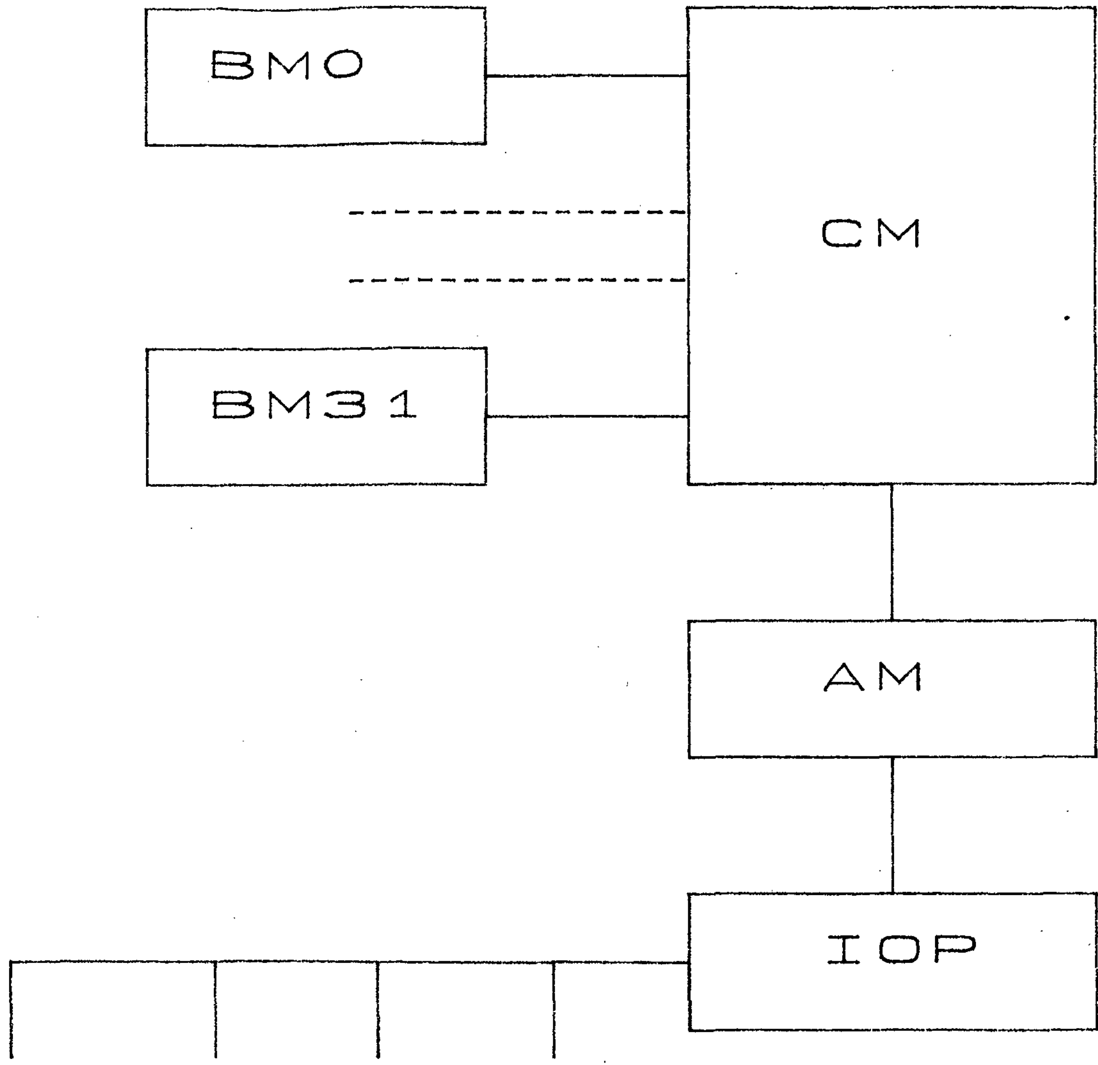


Figure A.1 A Typical Configuration of C-DOT DSS

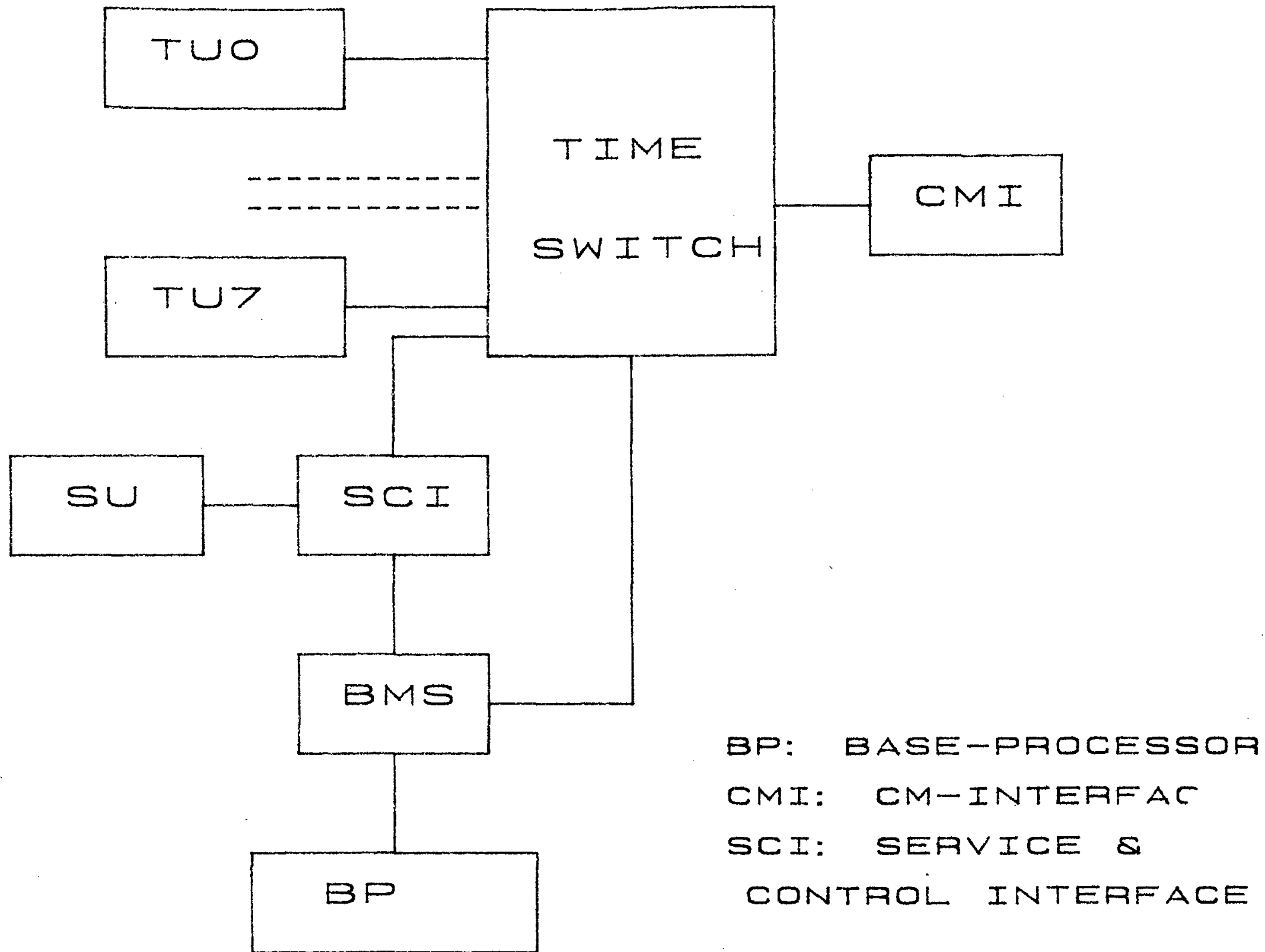


Figure A.2 A Typical BM Configuration

of interface equipment they contain. It performs the task of actual switching through various interfaces, like Dual Tone Multi Frequency receivers (DTMF), etc. on subscriber side. BMs are further modular in structure. A typical BM configuration is shown in Figure A.2. Base Processor (BP) provides the overall control. It is a 16/32 Bit 68010 CPU running under C-DOT REAL TIME OPERATING SYSTEM (CDOS). In its memory, 68000 has a large database for controlling its working. BM is connected to other links via high speed data links (HDLC). A detailed description of BM can be seen in "C-DOT DSS HARDWARE ARCHITECTURE" [24].

Administrative Module

This module provides administrative support to DSS for administrative and maintenance functions such as support for maintaining billing records, traffic information. The functions performed by AM include call processing functions, software recovery, overall initialization. It also provides interfaces to mass memory and operator terminals via IOP. It receives billing data from BMs on an hourly bases, and passes it to IOP.

Central Module

This provides the interconnection facility for BMs. It also has a message switch (MS) which handles the communication between BM and AM, BM and CM and between BMs (inter BM connection). Interconnections are provided through high speed data links. Different types of Links (depending upon speed and number of channels required) connect different modules. The architecture of CM is very much close to BM. CM comes into the picture only when three or more BMs are to be linked together. In the exchange with one/two BMs, functions of CM are performed by the BM/one of the BM. From IOP side both modules are equivalent.

Input Output Processor

IOP is a full fledged 16/32 Bit computer system with 68010 CPU running UNIX. IOP communicates with BM, via CM for various administrative and maintenance functions, and also supports a variety of peripherals such as printer, disk, magnetic cartridge tape, VDU etc. It can support a maximum of sixteen terminals. It acts as a front end processor for C-DOT DSS. The main functions performed by IOP are:

- Down loading software for DSS
- Handling databases for traffic and billing data
- Printing of billing and other reports
- Providing man machine interface for various maintenance and administrative operations

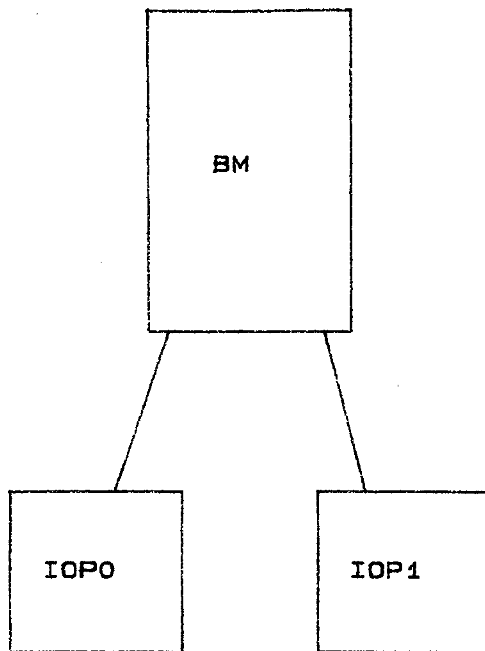


Figure A.3(a) Single Base Module

Configuration

(DUPLEX IOPs)

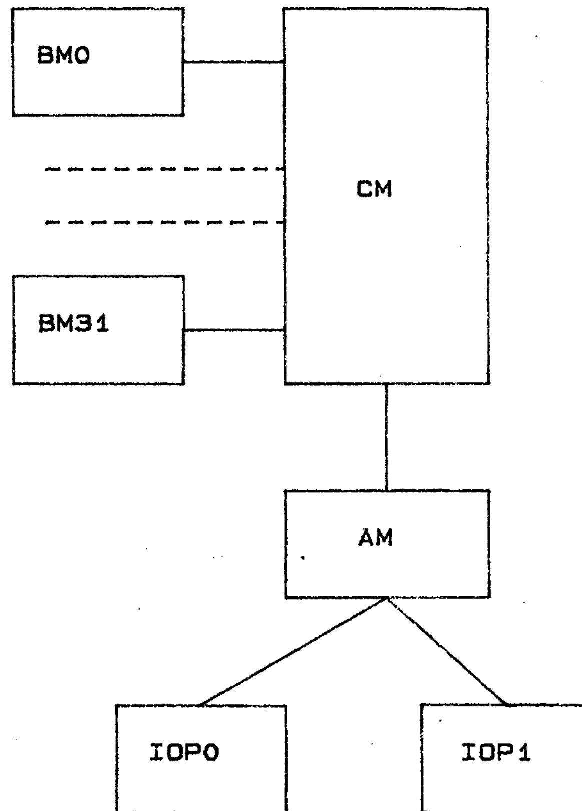


Figure A.3(b) Multi Base Module

Configuration

(DUPLEX IOPs)

- Fault detection and recovery
- System status display

The bus architecture is similar in characteristics to VME bus. IOP is also duplicated. Each copy is connected to other through High level Data Link Controller (HDLC) link. Two separate sets of I/O devices are attached to each IOP. I/O devices of each IOP are not duplicated. Hence each IOP with its I/O devices forms a security block. These IOPs are not configured as active/stand-by. Both IOPs are active at any given instant, one of them acting as master. All slave IOP - AP me ssages are routed through master IOP. Backup devices, like winchester are updated in both the IOPs; that is whenever an updation is performed on its winchester the master sends a message to other copy of IOP to perform a similar updation on its winchester.

Switching System Configuration

Using the above mentioned modules different types of switching systems can be designed. These configurations may have one IOP (simplex), which does not provide fault tolerance, or two IOPs (Duplex) which provides for fault tolerance.

The configuration which are of interest, with duplex IOPs are

Single Base Module (SBM)

In such a configuration only one BM is connected to IOP. No AM or CM is present in such a configuration. The functions of CM are performed by BM. Figure A.3(a) shows a single module configuration.

Multi Base Module (MBM)

In such a configuration more than one BM (up to 32) are connected via CM to AM and IOP. Every BM is assigned a unique number (from 0 to 31) which acts as BM identifier. All files corresponding to a given BM have BM number suffixed in their file name. BM number is also required while sending a message through HDLC. Such a configuration is shown in Figure A.3(b).

B.2 Software Architecture Overview

The software architecture in C-DOT DSS is distributed layered and highly modular in nature. Every layer present higher level of abstraction to layer above it. The software is

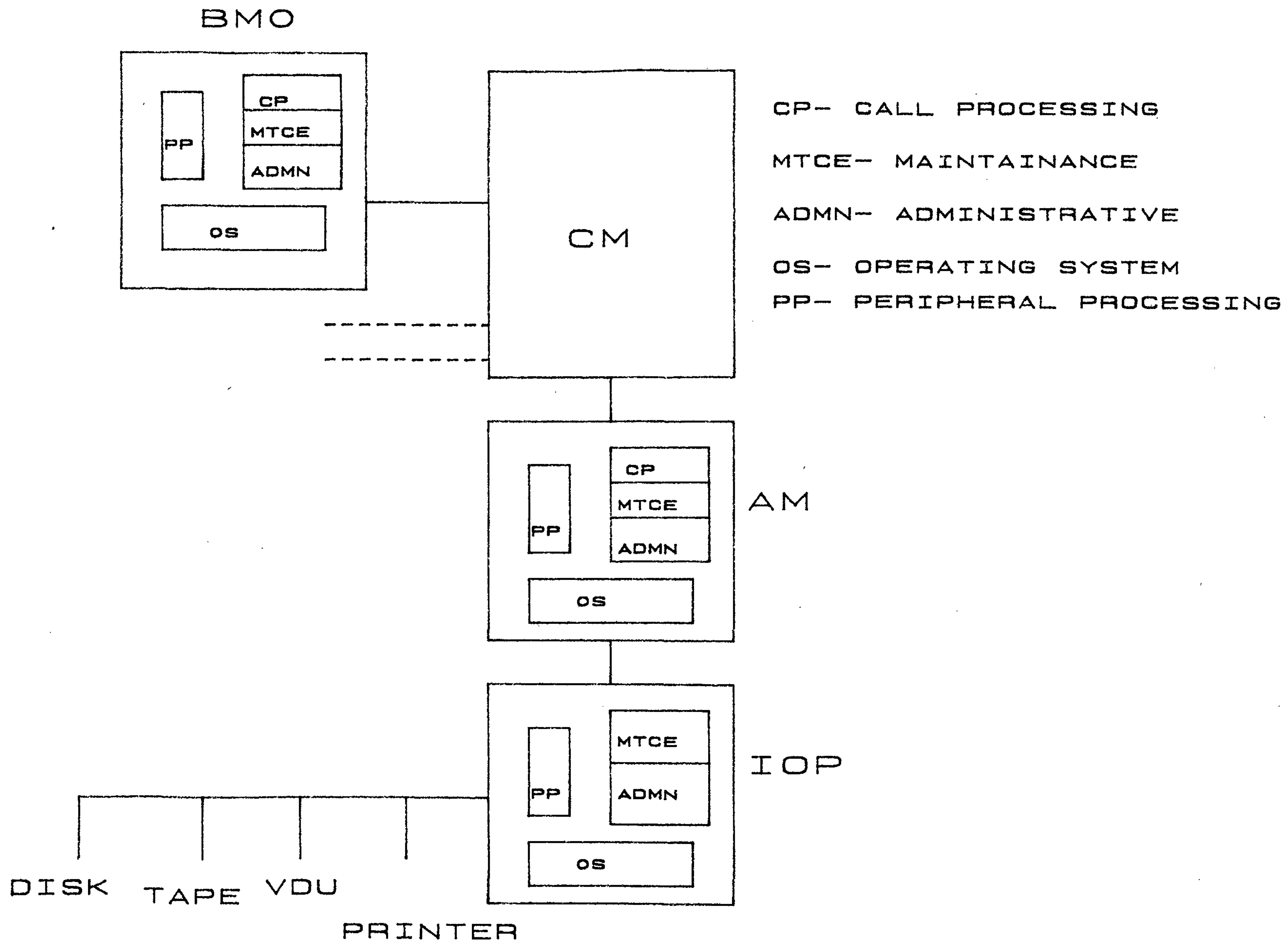


Figure A.4 Software Organization

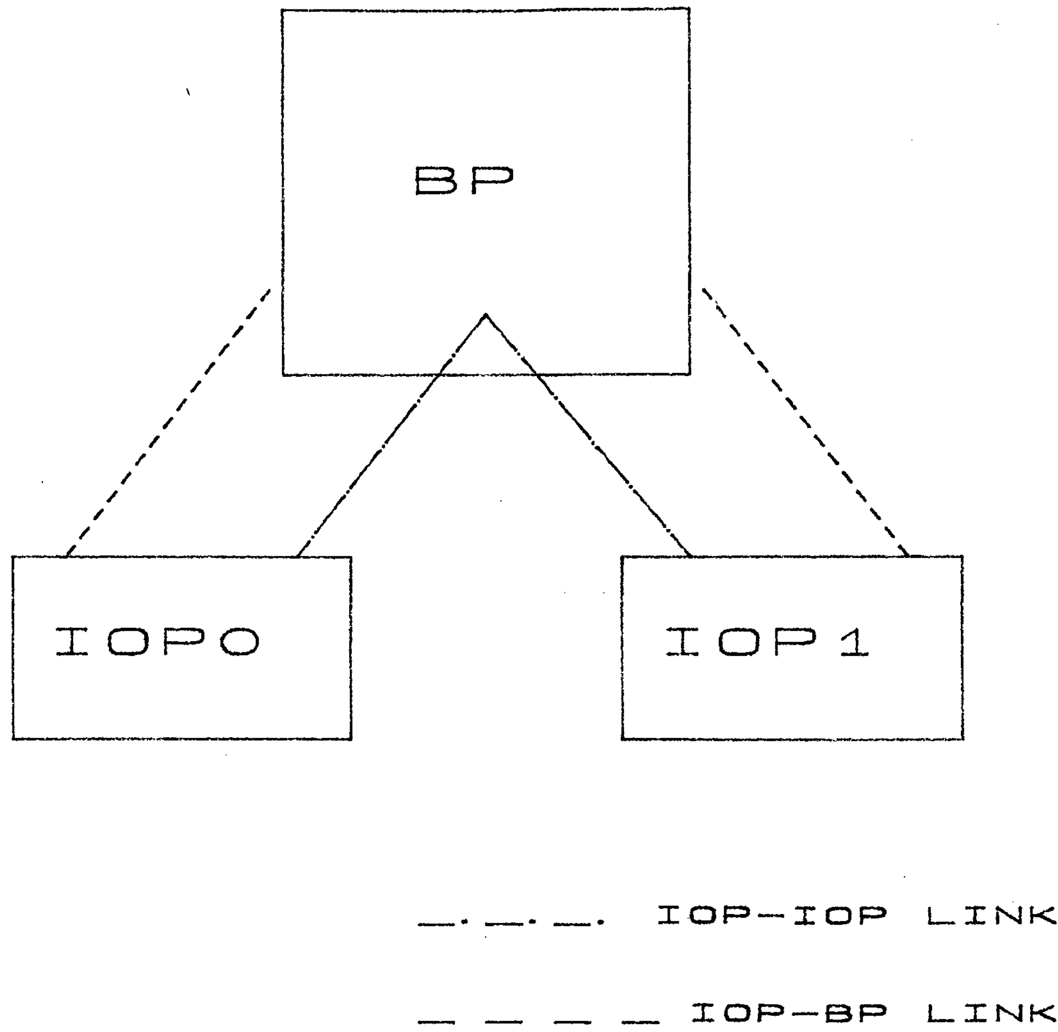


Figure A.5 HDLC Links for Communication

divided into a number of sub-systems. Figure A.4 shows the software architecture. Each sub-system consists of number of modules (called processes) and every module in turn a number of functions. The main sub- systems are

— CDOS - A real time operating system which provides uniform interface to application programs. In distributed architecture of C-DOT DSS, one of the important roles played by the CDOS is to provide inter process communication between process residing in the same or different processors, through HDLC links. BM runs under CDOS.

— Call processing sub-system - is responsible for executing functions which actually process a call, e.g. call routing, call metering etc.

— Administrative sub-system - provides for management of exchange (billing, operator commands etc.)

— Maintenance sub-system - provides functions for uninterrupted services to subscribers. It also provides for close monitoring of the systems sanity, comprehensive resource and data auditing facility which enables it to quickly detect faults and prevent their propagation.

— Data base sub-system - manages the data bases globally. It uses a sequential indexed file management system (C-ISAM). It hides the physical organization of data from application programs.

— Peripheral processors sub-system - Actual telephony hardware (with 6502 CPU) is controlled by the sub-system.

Inter Process Communication

Since different modules of different subsystems are functioning on different processors (in order to integrate them) interprocess (or interprocessor) communication becomes a vital aspect of software architecture. CDOS provides a uniform interface to all applications, so it becomes easy to communicate via CDOS. Figure A.5 shows how communication is performed between mate IOPs and between IOP and BP. (Here one point should be made clear, in SBM configuration IOP is connected to BM but in MBM it is connected to CM. Due to a likeness of CM and BM, for our purpose, we will not distinguish between CM and BM). High level data Link Controller (HDLC) is used to control the communication, and a comprehensive library of functions is provided for application programs. Communications within IOP, however, are handled via different means provided by UNIX; such as message queues, semaphores etc.

Every process has a UNIX message queue associated with it, by means of a unique

key, defined in C-DOT header files. All messages are to be received in this queue only. For sending messages the queue associated with destination process is used. HDLC also makes use of these queues for sending messages across the IOPs. This integrates the process of receiving a message from within the IOP and across the IOP.

File Management System

Although Unix offers only one kind of file, C-DOT DSS has two types of file systems. As provided by UNIX it has a distinct feature, UNIX file is a byte stream without any control character ! Practically it is possible to store every character (from 0 to 255) in a UNIX file. A main drawback in UNIX file is that it can not be truncated according to need. Only truncation allowed is of, zero length. This makes, removing of a few bytes from a UNIX file a complicated task. Different functions are provided by UNIX to manage this kind of file. The other file system supported by C-DOT is C-ISAM files. Built over UNIX file system, this type of file offers functions to store, retrieve and manipulate data in indexed sequential manner, and makes management of Database an easy task. Every C-ISAM file can have multiple indices with duplicate or unique key values. In C-DOT C-ISAM files are indexed on two four byte long fields with unique key value. For managing C-ISAM files different functions are provided in its library. However, there is no equivalent of lseek system call in C-ISAM file. To move around in C-ISAM file is very inefficient. Appendix F provides a list of C-ISAM library calls used in audit process. The information about record size, index fields, number of indices etc. is stored in the file itself and can be extracted from there. A B+ tree is maintained for sequential access of records in either increasing or decreasing order. It also provides for locking at record as well as file level. The information about locks is stored in a different file.

Processes On IOP

A number of processes are running on IOP. At any given instant both IOPs are active. The distinction between the IOPs is made on the basis of functions they perform. One of them is known as master, as it performs major functions and also directs its mate on other IOP (slave IOP) to perform the supporting functions. On both IOPs same executable image of a process is kept. Every program is divided in two distinct parts. After finding the IOP master/slave status, the program executes the required portion of program. As a number of process present in IOP is large, some processes have been added to schedule other related process to streamline the inter process communication, to ensure that mutually exclusive

functions are not executed simultaneously and to execute different functions according their priority.(for example Input Output Configuration Manager (IOCM))

B.3 Development Environment

The development environment at C-DOT is highly systematic. A number of VAXs, microVAXs (under VMS) and Motorola 68010 based systems (under UNIX) are used. These systems are interconnected via Ethernet. Motorola systems (commonly known as IOPs) provide an ideal environment, which is very close to target environment, for maintenance and administrative sub-systems. However, actual validation of software is done in laboratory, which has experimental DSS configurations, and on site at Delhi Cantt. and Ulsoor, Bangalore telephone exchanges. C programming language is used.

C-DOT has its own data structures for various purposes, defined in various "C" header files. Files and libraries are well organized and follow hierachial system provided by most operating systems. An application programmer is required to include these header files, depending upon configuration of DSS and different release. Further, files are placed in different directories [9] depending upon various conditions such as which subsystem they are meant for, whether a file is process related or data related etc.

Appendix C

Notes on Serial Communication

The Serial port on MS-DOS systems is capable of supporting the RS-232C standards for asynchronous communication. This Appendix explains on how to access the IBM PC hardware for serial communication. Serial, in contrast to parallel communication breaks a data item into its constituent parts, then transmits each part separately. On the receiving end, the parts reconstituted by reversing the process. Specifically, a 'parallel' byte is disassembled into its individual bits, which are then transmitted serially over the data link. At the receiver end the bits are once again assembled into byte. In fact, the 8250 Universal Asynchronous Receiver Transmitter (UART) performs precisely this function, as we will see later in this Appendix.

We explain some of the terms that have been used in the manuscript and as well they will be used here. The designers of ASCII were careful to limit the code to 7 bits so that the eighth bit could be employed for parity error checking after the 7 bits of a ASCII character are transmitted, an eighth, or parity, bit is transmitted whose value depends on whether the sum of the preceding seven bits was odd or even. Non-ASCII data, however, preserve the information coded into the eight bit and is transmitted as 8 bits with no parity bit. When the stream of bits is spewed onto the serial line, there is no accompanying signal to synchronize the receiver and the transmitter and hence the term asynchronous. For this reason, to enable the receiver to identify which bits constitute a byte, a logical 0 start bit is transmitted. Similarly, after all the bits have been transmitted, a logical 1 stop bit is transmitted. The receiver relies

on these framing bits to identify valid bit groups (byte) for reassemble. The speed at which bits are transmitted is called the Baud rate, i.e., the speed at which the communications line reverses its electrical polarity. Hence from the above we can see that the number of transmission bits for a byte is ten. The format of a single byte in asynchronous serial communication is shown in [1], [3], [4] and [5].

The UART has seven registers. We will see how these register can be used to make serial communication. The UART fetches bytes for transmission from the system's 8-bit parallel data bus into a transmission holding register where they are shifted onto the serial line at the current Baud rate. And conversely, the UART fetches the incoming bits, assembles them into a byte, which is then moved into a buffer register. This buffer enables the UART to begin assembly of the next incoming byte. This byte can be read from the buffer register, i.e., the UART will place the byte in the 8-bit data bus.

Clearly, the PC's processor can write bytes to the UART faster than the UART can disassemble and transmit them serially. Similarly, the processor can digest incoming bytes much faster than the UART can assemble them. To prevent the garbling of data that would result from overwhelming the transmitter or prematurely reading or re-reading the non-existent received byte, the UART contains a status register two bits of which reflect the readiness of its receiver and transmitter. By reading this register periodically, the process can know about the arrival of a byte or when the previous outbound byte was serialized and if the UART is ready for another. Since all UART functions are controlled by simple IN and OUT instruction, it should be convenient to write functions that enable to write and read the ports.

Communications with the 8250 UART takes place through seven registers located at processor's consecutive data port addresses. The PC supports serial cards known logically as COM1-4. The base port address of the UART for each of these devices is maintained in the ROM BIOS communications area beginning at 0:400-401H. The address of the first register of COM1 and COM2 are located at 0:400-401H and 0:402-403H respectively. The other registers pertaining to the COM ports can be had by indexing from their respective base address. The address of both transmitter and receiver are the same as the UART's base address, while the status register is at base address plus five (index of the status register

is five). Similarly, address of other registers can be calculated.

Most of the serialization error are shown by the serial state register. Meaning that the byte just received is not valid. Following are the bytes that represent different errors.

- bit 1 Overrun error.
- bit 2 Parity error.
- bit 3 Framing error.
- bit 4 Break.
- bit 5 Byte being received.
- bit 6 Byte being transmitted.
- bit 7 Unused.

The other registers are Baud rate register, Data format register and RS-232 output control register. The indices for these registers are 0, 1, 3, 4 respectively.

Appendix D

Notes on Interrupts

Interrupts are the natural way to handle intermittent request for attention. Most of the interrupts are required to identify itself to the microprocessor. It may identify itself explicitly in the form of the address of the service routine, or indirectly, as a code that the processor some how translates into the address of the service routine. Once the processor has discovered this address, it executes the code at the address, then resumes execution of the process from where it left off. This Appendix will discuss the IBM PC's hardware interrupts.

The processor comes to knows about the identity of an interrupt indirectly in the form of an interrupt identity, 0 through 255. This identity number acts as an index to an array whose each element contain the address an interrupt service routine (ISR). This table occupies the first 0x400 bytes of RAM-from 0000:0000 to 0000:03FF. Since each address is four bytes long, the vector corresponding to an interrupt can be found by multiplying the interrupt identity by four. And at this location we can find the address of the ISR. By resetting the interrupt flag in the processor's flag register we can mask any hardware interrupt service request. Setting this flag will allow the the processor to service the interrupts request.

D.1 Types of interrupts

The PC supports three different types of interrupts. These are Software interrupts, Hardware interrupts, and Predefined interrupts.

Software interrupts

The software interrupts are generated by the processor itself by executing an INT instruction. The software interrupts are not affected by the status of the interrupts flag.

Hardware interrupts

Hardware interrupts are generated by other physical device within the system. The request for hardware interrupts arrive via a pin in the processor chip. Note that this interrupts can be masked.

Predefined interrupts

The first five interrupts are usurped by the processor for internal use. For example, he interrupts type 0 is for trapping division by zero, and interrupt type 2, known as nonmaskable interrupts NMI, is used to bring the system to a halt in case of memory parity error.

D.2 Hardware Interrupts

We, in this Appendix, will consider only the hardware interrupts. And the interrupts related to serial communication falls under this category. The eight interrupts lines on the PC's system bus are termed as IRQ0 to IRQ7. Only two of these interrupts are permanently dedicated-the system timer is hardwired to IRQ 0 and the keyboard to IRQ 1. The remaining IRQs are assigned arbitrarily by the Peripheral Interrupt Controller. These IRQ lines do not lead directly to the processor, but to the Intel 8259 Peripheral Interrupts Controller (PIC), which acts as a receptionist to the processor by controlling the interrupt traffic. When any of the peripheral devices places its request on its assigned IRQ line, the PIC first examines its interrupts mask register to ascertain whether the system is entrusted in an interrupt from that device. A TRUE bit (the appropriate bit) in the interrupt mask register of the PIC indicates that the interrupt is masked, i.e., not active. On the other hand, if the appropriate bit in the interrupt mask register is FALSE, the interrupt is not masked and the PIC signals the processor that an interrupts has occurred. When the processor finishes its current instruction, it checks the interrupt flag; if it is FALSE it ignores the PIC and if the interrupt flag bit is TRUE, the processor signals an acknowledgment to the PIC, which inturn responds by placing on the data bus the number of the IRQ line that generated the interrupt request plus eight, which in fact gives the interrupt identity. The eight hardware interrupt are given below with their

corresponding interrupts request line.

IRQ0	Int 8	System timer
IRQ1	Int 9	Keyboard
IRQ2	Int A	reserved
IRQ3	Int B	Asynchronous communication (COM1)
IRQ4	Int C	Asynchronous communication (COM2)
IRQ5	Int D	Fixed disk
IRQ6	Int E	Diskette
IRQ7	Int F	Printer

To process hardware interrupts correctly it is essential that we mask and unmask the appropriate the bits in the controller's interrupts mask register (IMR). This read-write register is located at the port address 21H.

Interrupt priority is also maintained. As initialized by the ROM BIOS, the PIC is placed in the fully nested priority mode. This means that lower-numbered interrupts are always serviced before ones with higher numbers. It is required that the hardware interrupt handlers inform the PIC when an interrupt service is complete. In this mode it is not necessary to identify the interrupt type that has just completed, so this command is called a non-specific end-of-interrupt (EOI) command. The EOI command sequence is

```
mov   al, 20h
out   20h, al
```

Note that this should appear in all hardware interrupts and should be placed as close as possible to the IRET instruction.

Appendix E

Notes on the IBM EGA

The Enhanced Graphic Adapter (EGA) is rapidly becoming the most common graphic card in the MS-DOS world.

There are four different graphic standards supported by the EGA.

— Color Graphic Adapter (CGA) compatible graphic mode.

— EGA graphic mode for 200-lines color monitors.

— EGA graphic mode for 350-lines color monitors.

— EGA graphic mode for use with monochrome (text) monitors.

The original EGA from the IBM comes with 64K of graphic display memory on the card. This may be expanded on increments of 64K to 256K. The more the EGA memory, greater the graphic capabilities. The EGA is designed to work with one of three different monitors: the IBM Color Display, the IBM Enhanced Color Display (ECD), or the IBM Monochrome Display.

In this appendix we will consider only ECD monitor. This monitor is compatible with all the modes used with the Color Display monitor, and uses one more high resolution mode. This high resolution mode 16, can be used only with the IBM ECD or its equivalents.

The EGA can display 16 colors from a 64-color palette when used with the ECD. The 16 colors are available only in mode 16 if there is more than 64K on the EGA. In high resolution mode, EGA can maintain upto two pages of Graphic starting at segment 0xA000. But again

this depends on the amount of memory installed on the card.

The information about the presence of EGA card, its memory and the type of the monitor can be had from the encoded information in the BIOS Data Area in the byte 0x40:0x87. It is one of the several status bytes kept by the EGA BIOS for its internal use and to provide information to the programs (note that EGA has a new BIOS that replaces the original PC video functions and adds several new functions).

E.1 Display Memory Organization

The EGA has two different display memory organizations for graphics. In modes 4 through 6, the EGA uses the same memory organization as the CGA. In those modes the display memory segment starts at 0xB800 and uses 80 bytes per scan line. The display memory for modes 13 through 16 start at the segment 0xA000 and uses upto 64K of the 80x86 CPU address space. Each byte represent 8 pixel, with the most significant bit being the leftmost. The scan lines are not separated like they are in the CGA modes, so the byte offset of a pixel is easier to calculate. In mode 16, the EGA has a maximum resolution of 640 x 350, or 224,000 pixels. Since there are upto 16 color, each pixel must use 4 bits to specify the color. Altogether, this represents a total memory usage of 109K.

The 80x86 CPU used in the PC can address only a segment of 64K. The EGA fits into the 64K segment limit by dividing 128K of its 256K memory into four 32K bit planes. Each bit plane corresponds to one bit of a pixel color. Imagine these four bit planes as being stacked on top of each other at the same CPU address.

Reading or Writing 4 different bytes (one for each bit plane) at the CPU address presents a problem. To overcome this problem, the EGA has four latch registers. The latch registers temporarily hold one byte from each of the bit plane. The EGA logic fills each of the four bit planes at the address last read by the CPU. When the CPU sends a byte to the address last read, each of the four latch registers may be unchanged, modified, or entirely replaced by the CPU data. The latch register contents are then written back to each of the EGA's bit planes. When the latch registers are written back to the EGA's bit planes, they are

again "stacked" with one bit of each 4 bytes forming the 4-bit color for 8 pixel.

E.2 Bit Mask Register

It is important to note that the byte returned to the CPU after reading an address in the EGA display memory has no use. That byte is read only to establish which pixels to work with and to "prime" the latch registers, allowing the individual bytes of the bit plane to be manipulated by the CPU data. Whether the latch register are modified, replaced, or unchanged by the CPU depends on the setting of several EGA control registers. These registers are accessed through one of five indexed VLSI chips on the EGA. These VLSI chips are set by sending an index number corresponding to the function desired, followed by the data for that function.

For example, the EGA has a bit mask register that will allow individual bits to the latch register to be protected from change. Setting a bit to 0 in this register masks out the corresponding bit in the latch register, and setting a bit to 1 allows that bit to be changed by the CPU writes. This is programed by sending an index 8 (function number 8) to the port 0x3CE, followed by the bit mask register to the port 0x3CF.

E.3 Map Mask Register

A second register that affects how the latch register contents are re-written is the map mask register. If any of the four bits of the map mask register are zero, the corresponding bit maps (bit planes) are protected from change. Sending a number between 0 and 15 to the map mask register will allow the color corresponding to that number to be written to the EGA's bit plane. However, note that the previous contents of the bit map should be cleared before setting the map mask to mask for a new color; but after setting the bit mask, by writing a zero to the byte containing the pixel to change. The map mask register is part of the EGA's Sequencer Chip. The function number is 2 (index is 2) and this should be sent to the port 0x3C4 and the map mask should be sent to the port 0x3C5.

E.4 Set/Reset Register

There is another register available with the EGA called the set/reset register. The

set/reset register will set a byte to 0xFF in each EGA bit plane where a bit on in the set/reset regsiter, and will reset a byte to 0 in each EGA bit plane where a bit is off. Therefore, the previous contents of the latch register are replaced with the color number corresponding to the value in the set/reset register. The map mask register has no effect on the set/reset register, but the bit mask register is usable to protect adjacent pixels. To use the set/reset register, one must first enable it with the enable set/reset register. The set/reset register and the enable set/reset register are part of the EGA's graphic controller chip. The set/reset register is accessed by first sending an index 0 to the port 0x3CE and then sending the four-bit color code to the port 0x3CF. The enable set/reset register can be accessed by index 1 to the same port as above and also the data port remains the same.

E.5 EGA Write Modes

The EGA has three write modes: 0, 1, and 2. Changing the write mode changes the way that EGA hardware reacts when the CPU sends a byte to the display buffer. Each write mode is optimized for a different use. Write mode 0 is the general purpose write mode, write mode is optimized for copying EGA memory regions, and write mode 2 is best used for color fills. To access this function send index 5 to the port 0x3CE and the data to the port 0x3CF.

E.6 EGA Color Palettes

The 64-color palette has the same three basic color (red, green, blue) as the 16-color palette, but there is no intensity bit. Instead, each color has 2 bits for individual color intensity, giving three intensity levels for each color. The total 64-color palette may thus be represented with 6 bits (3 color x 2 bits). The bits for the lower intensity of the three colors are the most significant bits in the 6-bit value. The least significant 3 bits represent the higher-intensity red, and blue. EGA palette registers are write only. When changing the palette through the EGA BIOS function call will check for the existence of a 256-byte table called the parameter save area when changing the palette register. BIOS will save the six bit color if the table exists. If the BIOS is not called to change the palette then one must update the parameter

table himself.

E.7 Data Rotate Register

The data rotate register allows one to select how the data sent by the CPU will be combined with the EGA latch register. The options are to have the data be ANDed, ORed, XORed, or unmodified with the bytes in the latch registers. Although the data rotate register also has the ability to rotate the data coming from the CPU, in practice this is of little use. The index for this function is 3. The ports for the function number and data remain the same as above.

There are many more features available with the EGA. For details see [1], [15], and [20].

Appendix F

System And Function Calls

This appendix explains about the system call and function call used in the source code. All the different types of call have been put under different headings. For further details the reader is referred to [26]-[28] for UNIX system calls and function call. For the C-ISAM function call the reader is referred to [10].

F.1 UNIX System Calls

- int creat (*filename, mode)

Creats a file or prepares to rewrite an already existing file by name pointed by *filename. Returns filedescriptor or -1.

- int open (*filename, mode)

Opens a file pointed by filename with mode specifying read only, write only etc. Returns file descriptor or -1.

- int close (fd)

Closes an open file described by filedescriptor. Returns 0 or -1.

- int read (fd, *buffer, nbytes)

Reads nbytes number of bytes in a buffer pointed by *buffer from a file described by filedescriptor fd. Returns number of bytes actually read or -1.

- int write (fd, *buffer, nbytes)

Writes nbytes number of bytes from a buffer pointed by *buffer onto a file described

by filedescriptor fd. Returns number of bytes actually written or -1.

- int lseek (fd, nbytes, position)

Moves the file pointer of file described by fd by nbytes number of bytes from a position specified by position. If position = 0 then start of file is taken, position = 1 then current position of file is taken, position = 2 end of file is taken. nbytes can be negative or positive. Returns position of file pointer from start of file.

- int stat (*filename, *buffer)

This call fills the buffer pointed by *buffer with information about the file pointed by *filename (such as file size, date of creation etc.). Returns 0 or -1.

F.2 'C' Function Calls

- char *getenv(*name)

This function finds the value associated with shell environment variable 'name'. Returns pointer to value or NULL.

char *malloc(size)

Allocates a memory area for the calling process of size 'size'. Returns pointer to memory area or NULL.

- void perror (*message)

Displays description of error encountered by calling process alongwith message pointed by 'message'.

- int fprintf(*ptr, control, arg1, arg2, ...)

Prints arguments given in the list onto a file pointed by ptr according to a format specified by control. Returns number of characters printed or -1.

- int printf (control, arg1, arg2, ...)

Same as fprintf. Only difference is that printing is performed on standard output file (stdout).

- int scanf (control, arg1, arg2, ...)

Scans the standard input file (stdin) and accepts the values for arguments according to format as specified by control. Returns number of arguments successfully read. If input terminates before any kind of conflict occurs then it returns EOF.

- char *strcpy (*s1, *s2)

Copies string pointed by s2 on string pointed by s1 stoping after null byte has been copied. Returns pointer to s1.

- char *strcat (*s1, *s2)

Concatenates string s2 with string s1. The result is null terminated string. Returns pointer to s1.

- int system (*command)

Issues a shell command pointed by 'command'. Returns -1 if it is unable to issue the command. However if command is not a legal shell command, no error is reported.

F.3 C-ISAM Function Calls

- int isopen (*name, mode)

Opens a C-ISAM file in the mode specified by 'mode'. Returns file descriptor of the file or -1.

- int isclose (fd)

Closes an already open C-ISAM file. Returns file descriptor of closed file or -1.

- int isread (fd, key, *buffer, mode)

Reads a record from a C-ISAM file described by 'fd' into buffer pointed by 'buffer'. Mode specifies wheather next or previous or first or last or current record is to be read or a recrd is to be read according to a specified key. Returns 0 or -1.

- int isrewritecurr (fd, *buffer)

Rewrites current record of a C-ISAM file described by 'fd' from a buffer pointed by 'buffer'. Returns 0 or -1.

- int iswrite (fd, *buffer)

Writes (adds) a record in a C-ISAM file from a buffer pointed by 'buffer'.

Returns 0 or -1.

- int isdelcurr (fd)

Deletes the current record in a C-ISAM file described by 'fd'. Returns 0 or -1.

- int isindexinfo (fd, *buffer, type)

Fills the buffer pointed by *buffer with information about index of a C-ISAM file described by 'fd'. The index information is of two types. 'type' specifies what kind of

information is required. Returns 0 or -1.

- int isstart (fd, key, mode)

Brings the file pointer of a C-ISAM file described by 'fd' at the specified position. Position can be first, last or as specified by key. Returns 0 or - 1.

Appendix G

Bibliography

[1] **Angermeyer, J.**, et.el., "The Waite Group's MS DOS Developer's Guide," Indianapolis, Indiana: Howard W. Sams & Company, Inc., 1988.

[2] **Bach, M.J.**, "The Design of UNIX Operating System," Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1986.

[3] **Campbell, J.**, "A Programmer's Guide to Serial Communication," Indianapolis, Indiana: Howard W. Sams & Company, Inc., 1986.

[4] **Campbell, J.**, "The RS-232 Solution," Berkeley, California: Sybex, Inc., 1984.

[5] **Campbell, J.**, "Crafting C Tools for the IBM PC," Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1986.

[6] **C-DOT**, "C-DOT DSS Training, Lecture Notes on IOP & UNIX," New Delhi, New Delhi: C-DOT Publication, 1986.

[7] **C-DOT**,"C-DOT DSS Training, Lecture Notes on Software Architecture Overview," New Delhi, New Delhi: C-DOT Publication, 1986.

[8] **C-DOT**, "C-DOT 512 MAX General Description," New Delhi, New Delhi: C-DOT Publication, 1986.

[9] **C-DOT**, "C-DOT DSS Training, Lecture Notes on Central File System," New Delhi, New Delhi: C-DOT publication, 1986.

[10] **C-ISAM**, "Reference Manual," Palo Alto, California: Relational Database Systems, Inc., 1982-85.

[11] **Christain, K.**, "The UNIX Operating System," New York, New York: John Wiley & Sons, Inc., 1983.

[12] **Duncan, R.**, "Advanced MS DOS," Redmond, Washington: Microsoft Press, 1986.

[13] **IBM**, "Disk Operating System 3.00 Technical Reference," Boca Raton, Florida: IBM Corporation, 1984.

[14] **IBM**, "Personal Computer Technical Reference," Boca Raton, Florida: IBM Corporation, 1983.

[15] **IBM**, "IBM Enhanced Color Display," Boca Raton, Florida: IBM Corporation, 1986.

[16] **Johnson, N.**, "Advanced Graphics in C: Programming and Techniques," Berkeley, California: Osborne Mc Graw- Hill, 1987.

[17] **Kernihgan, B.W., and Dennis, M.R.**, "The C Programming Language," Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1987.

[18] **Kernihgan, B.W., and Pike, R.**, "The UNIX Programming Environment," Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1984.

[19] **King, R.A.**, "The IBM PC-DOS Handbook," Berkeley, California: Sybex, Inc., 1983.

[20] **Kliewer, B.D.**, "EGA/VGA A Programmer's Reference Guide," New York,New York: McGraw-Hill Book Company, 1988.

[21] **Knuth, D.E**, "Fundamental Algorithms: The Art of Programming Vol. 1," Reading, Mass.: Addison-Wesley Publishing Company, Inc., 1973.

[22] **Knuth, D.E.**, "Seminumerical Algorithms: The Art of Programming Vol. 2," Reading, Mass.: Addison-Wesley Publishing Company, Inc., 1973.

[23] **Lal, K.B., Chandrasekaran, T., Pandey, Y.K.**, "C-DOT DSS Architecture - Overview of C-DOT & DSS Projects," New Delhi, New Delhi: C-DOT Publication, October 1986.

[24] **Lal, K.B., Chandrasekaran, T., Pandey, Y.K.**,"C-DOT DSS Hardware Architecture - Overview of C-DOT & DSS Projects," New Delhi, New Delhi: C-DOT Publication, October 1986.

[25] **Lal, K.B.**, et. al., "C-DOT DSS Software Architecture - Overview of C-DOT & DSS Projects," New Delhi, New Delhi: C-DOT Publication, October 1986.

[26] **Motorola**, "System V/68 Programmer's Reference Manual," Tempe, Arizona: Motorola, Inc., 1985.

[27] **Motorola**, "System V/68 User's Reference Manual," Tempe, Arizona: Motorola, Inc., 1985.

[28] **Motorola**, "System V/68 Support Tools Guide," Tempe, Arizona: Motorola, Inc., 1985.

[29] **Norton, P.**, "Inside the IBM PC," Bowie, MD: Robert J. Brady Co., 1983.

[30] **Rochkind, M.J.**, "Advanced UNIX Programming," Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1985.

[31] **Rogers, D.F.**, "Procedural Elements for Computer Graphics," Singapore, Singapore: Mc Graw-Hill Book Company, 1985.

Appendix H

Source Listing

This appendix gives the source code written for the development of GIM. The reader may note that not all of the files have been listed here. Only the major files which give an overview of the implementation have been listed. This appendix have been divided into four section. The first two section gives the source listing of the two libraries. The third section deals with the GIM source code. Finally the last section gives a few snapshots of the GIM. For coding convention please refer to section 1.4.

H.1 CGRAPHIC Listing

graph.h

```
# ifndef _GRAPH
# define _GRAPH
# define C_REP( 0x00 < 3 )
# define C_AND( 0x01 < 3 )
# define C_OR( 0x02 < 3 )
# define C_XOR( 0x03 < 3 )
# define C_GRANC( 0x01 < 8 )
# define C_GRANF( 0x02 < 8 )
# define C_PSLPC( 0x03 < 8 )
# define C_NSLPF( 0x04 < 8 )
# define C_NSLPC( 0x05 < 8 )
# define C_PSLPF( 0x06 < 8 )
# define C_HTONE( 0x07 < 8 )
# define C_LIQUID( 0x08 < 8 )
# define C_CORRUG( 0x09 < 8 )
# define C_DOT( 0x10 < 8 )
# define LI_BLACK0x00
# define LI_BLUE0x01
# define LI_GREEN0x02
# define LI_RED0x04
# define LI_CYAN( LI_GREEN | LI_BLUE )
# define LI_MAGENTA( LI_RED | LI_BLUE )
# define LI_BROWN( LI_RED | LI_GREEN )
# define LI_WHITE( LI_RED | LI_GREEN | LI_BLUE )
# define HI_INTENSITY0x08
# define HI_BLACK( HI_INTENSITY )
# define HI_BLUE( HI_INTENSITY | LI_BLUE )
# define HI_GREEN( HI_INTENSITY | LI_GREEN )
# define HI_RED( HI_INTENSITY | LI_RED )
# define HI_CYAN( HI_INTENSITY | LI_CYAN )
# define HI_MAGENTA( HI_INTENSITY | LI_MAGENTA )
# define HI_BROWN( HI_INTENSITY | LI_BROWN )
# define HI_WHITE( HI_INTENSITY | LI_WHITE )
# define FILL_RTOL0
# define FILL_LTOR1
# define FILL_TTOB2
# define FILL_BTOT3
# define FILL_CRUSH4
# define FILL_EXPLD5
# define FILL_CONVH6
# define FILL_CONVW7
# define FILL_SPRDH8
# define FILL_SPRDW9
# define FILL_BTOP10
# define FILL_BBOT11
# define FILL_BLEFT12
# define FILL_BRIG13
# define FILL_BCEN14
```

```
# define FILL_BEDGE 15
# define FILL_RAND16
# define FRM_POS0x0f
# define FRM_TOP0x01
# define FRM_BOT0x02
# define FRM_LFT0x01
# define FRM_RGT0x02
# define FRM_CEN0x03
# define FRM_STY0xf0
# define FRM_BOX0x10
# define FRM_BGR0x20
# define FRM_UND0x30
# define CURS_ARROWNE0
# define CURS_ARROWNW1
# define CURS_CROSSWIRE2
# define CURS_CROSSHAIR3
# define CURS_BUTTERFLY4
# define CURS_JUMP0
# define CURS_FAST1
# define CURS_MED2
# define CURS_SLOW3
# define HIDE_ARRO(0x01 < 0)
# define SHOW_GRID(0x01 < 1)
# define HIDE_CALI(0x01 < 2)
# define HIDE_POSX(0x01 < 3)
# define HIDE_POSY(0x01 < 4)
# define HIDE_NEGX(0x01 < 5)
# define HIDE_NEGY(0x01 < 6)
# define HIDE_X(HIDE_POSX | HIDE_NEGX)
# define HIDE_Y(HIDE_POSY | HIDE_NEGY)
# define HIDE_NXNY(HIDE_NEGX | HIDE_NEGY)
# define HIDE_AXIS(HIDE_X | HIDE_Y)
# define HIDE_BOXS(0x01 < 7)
# define HIDE_BCOL(0x01 < 8)
# define CHAR_XTOP(0x01 < 9)
# define CHAR_YRIG(0x01 < 10)
# define CHAR_XCOL(0x01 < 11)
# define CHAR_YCOL(0x01 < 12)
# define BAR_YAXIS(0x01 < 0)
# define BAR_RANGE(0x01 < 1)
# define BAR_DIVDE(0x01 < 2)
# define BAR_NOBOX(0x01 < 3)
# define BAR_SHADW(0x01 < 4)
# define BAR_CLIPT(0x01 < 5)
# define BAR_TOPCH(0x01 < 6)
# define EL_FULL0xffffffffL
# define EL_DIT10xaaaaaaaaL
# define EL_DIT20x88888888L
# define EL_ENGG0xfc01803fL
# define PT_NUL0
# define PT_DOT1
# define PT_SQR2
# define PT_BOX3
typedef struct {
 long ox, oy;
 long max_x, max_y;
 long min_x, min_y;
 int  tlx, tly;
 int  brx, bry;
} Mapping;
typedef struct {
 int     options;
 int     dx, dy;
 char    **p_x;
 char    **p_y;
 int     axiscolor, axismix;
 int     boxcolor, boxmix;
 int     gridcolor, gridmix;
 int     charcolor, charmix;
 int     backcolor, backmix;
} Calibration;
typedef struct {
 unsigned int options;
 int  barwidth;
 int  offset;
 int ndivd;
 int *divcolor;
 int  nbars, *ppts;
 int barcolor, barmix;
 int  boxcolor, boxmix;
 int  shadowcolor, shadowmix;
} Bar;
# endif
```

debug.h

```
# ifndef _DEBUG
# define _DEBUG
# define INVX 1
# define INVY 2
# define INVCOL3
# define CANTOPEN7/* Cannot open file */
# define CANTREAD8/* Cannot read */
# define CANTWRITE9/* Cannot write */
# define INVCRS10/* Invalid Cursor style */
# define INVCRN11/* Invalid Cursor number */
# define INVCRM12/* Invalid Cursor movement style */
# define INVPAGE13/* Invalid page no. */
# define INVMPN14/* invalid map number */
# define INVPTTYPE15/* Invalid point type */
# define INVMAPX16/* Invalid map x coordinate */
# define INVMAPY17/* Invalid map y coordinate */
# define INVOX18/* Invalid map origin x coor  */
# define INVOY19/* Invalid map origin y coor  */
# define HORLINE0
# define VERLINE1
# define BOXSHELL2
# define BOXFILL3
# define PRINTCHAR4
# define PRINTROW5
# define PRINTCOL6
# define SETGMODE7
# define SPREAD8
# define WRITECHAR9
# define WRITEROW10
# define WRITECOL11
# define PROWFRM12
# define PCOLFRM13
# define LINE  14
# define RESTORE15
# define SAVE16
# define CURSSET17
# define CURSMOVE18
# define CURSRESET19
# define SETPAGE20
# define DISPAGE21
# define GETSTR22
# define POINT23
# define PLOT24
# define MAP  25
# define BAR  26
# define CALIBRATE27
# define CONNECT28
# define ELINE29
extern int gperror;
extern int gfunction;
extern void setgperror(), resetgperror(), gerror();
# endif
```

debug.c

```
# ifdef DEBUG
# include "debug.h"
# include "graph.h"
# include "screen.h"
intgerrorno;
intgfunction;
static intgperror = 1;
static char *error_msg[] = {
 /* dummy 0 */ "",
 /* INVX   1 */ "Invalid x coordinate = %d",
 /* INVY   2 */ "Invalid y coordinate = %d",
 /* INVCOL  3 */ "Invalid color = %d",
 /* INVMIX  4 */ "Invalid mix = %d",
 /* INVMODE  5*/ "Invalid mode = %d",
 /* INVFILE  6 */"Invalid format %s",
 /* CANTOPEN7*/"Cannot open file %s",
 /* CANTREAD8*/"Read error on %s",
 /* CANTWRITE 9*/"Write error on %s",
 /* INVCRS  10*/"Invalid cursor style = %d",
 /* INVCRN  11*/"Invalid cursor no = %d",
 /* INVCRM 12*/"Invalid cursor move style = %d",
 /* INVPAGE   13*/"Invalid page no = %d",
 /* INVMPN  14*/"Invalid map no = %d",
 /* INVPTTYP 15*/"Invalid point type = %d",
 /* INVMAPX  16*/"Invalid map x coordinate = %d",
 /* INVMAPY  17*/"Invalid map y coordinate = %d",
 /* INVOX 18  */"Invalid map origin x = %d",
 /* INVOX 19  */"Invalid map origin y = %d"
};
static char *func_name[] = {
 "HORLINE"   /*0*/,
 "VERLINE"   /*1*/,
 "BOXSHELL" /*2*/,
 "BOXFILL"   /*3*/,
 "PRINTCHAR"/*4*/,
 "PRINTROW" /*5*/,
 "PRINTCOL" /*6*/,
 "SETGMODE"/*7*/,
 "SPREAD"   /*8*/,
 "WRITECHAR"/*9*/,
 "WRITEROW"/*10*/,
 "WRITECOL" /*11*/,
 "PROWFRM" /*12*/,
 "PCOLFRM"  /*13*/,
 "LINE"       /* 14*/,
 "RESTORE"  /*15*/,
 "SAVE"      /* 16*/,
 "CURSSET"  /*17*/,
 "CURSMOVE"/*18*/,
 "CURSRESET"/*19*/,
 "SETPAGE"  /*20*/,
 "DISPAGE"  /*21*/,
 "GETSTR"   /*22*/,
 "POINT"     /*23*/,
 "PLOT"      /* 24*/,
 "MAP"       /* 25*/,
 "BAR"       /* 26*/,
 "CALIBRATE"/*27*/,
 "CONNECT" /*28*/,
 "ELINE"     /*29*/
};
void
setgperror()
{
 gperror = 1;
}
void
resetgperror()
{
 gperror = 0;
}
void
gerror( errorno, value )
int errorno;
int value;
{
 gerrorno = errorno;
 if ( gperror ) {
   static char buffer1[ 40 ];
   static char buffer2[ 40 ];
   sprintf( buffer1, error_msg[ gerrorno ], value );
   sprintf( buffer2, "%s :: %s",
     func_name[ gfunction ], buffer1);
   spread( 0, YMAX - 12, XMAX, YMAX,
     LI_RED, C_REP, FILL_TTOB );
   printrow( 10, YMAX-9, LI_WHITE, C_REP, buffer2 );
   printrow( (XMAX > 1) + 10, YMAX-9, LI_BLACK, C_REP,
     "type 'a' to abort, 'p' to proceed");
   while ( 1 ) {
     char c = getch();
     if ( c == 'a' || c == 'A' ) {
       setgmode( 3 );
       exit(1);
     }
     if ( c == 'p' || c == 'P' ) {
       spread( 0, YMAX - 12, XMAX, YMAX,
         LI_RED, C_REP, FILL_TTOB );
       return;
     }
   }
 }
}
# endif
```

screen.h

```
# ifndef _SCREEN
# define _SCREEN
# define _EGA16
# ifdef _EGA16
# include "screen16.h"
# endif
# ifdef _EGA14
# include "screen14.h"
# endif
# define XMAX( N_ROW_PIXELS - 1 )
# define YMAX( N_COL_PIXELS - 1 )
# define CWIDTH( N_ROW_PIXELS / N_ROW_CHARS )
# define CHEIGHT( N_COL_PIXELS / N_COL_CHARS )
# define PAGESIZE((N_ROW_PIXELS / 8L) * N_COL_PIXELS * 4L)
# endif
```

screen16.h

```
# ifndef _SCREEN16
# define _SCREEN16
# define EGAMODE16
```

```
# define   N_ROW_PIXELS640
# define   N_COL_PIXELS350
# define   N_ROW_CHARS80
# define   N_COL_CHARS25
# define   WCHAR0x08
# define   HCHAR0x08
# endif
```

umband.c

```
void _multiband( xoff, nmasks, pmasks, sy, ly )
int xoff, nmasks; unsigned char *pmasks; int sy, ly;
{
 register int  n = ly - sy + nmasks;
 register int  i = nmasks;
 register char  far*base;
 while ( -- i >= 0 ) {
   if ( *(pmasks + i) == 0 )
     continue;
   SETMASK( *(pmasks + i) );
   base = BASE( xoff < 3, sy + i );
   {
     register intm;
     for ( m = (n - i) / nmasks ; m > 0 ; m -- ){
       MODIFY( base );
       base += nmasks * WIDTH;
     }
   }
 }
}
```

boxfill.c

```
int
boxfill( x1, y1, x2, y2, colour, mix )
int x1,y1,x2,y2,colour,mix;
{
 register int  sx, sy;
 register int  lx, ly;
# ifdef DEBUG
# include "debug.h"
 gfunction = BOXFILL;
 VALXY( x1, y1 );
 VALXY( x2, y2 );
 VALCOLMIX( colour, (unsigned char )mix );
 VALDITH( mix );
# endif
 MAXMIN( x1, x2, lx, sx );
 MAXMIN( y1, y2, ly, sy );
 _initega();
 sx ++; sy ++;
 lx --; ly --;
 SETCOLOR( colour, ( unsigned char )mix );
 {
   int  lxoff,sxoff;
   unsigned charmask;
   register struct mask *pmasks_tbl =
     _getmask( (char)(mix > 8) );
   sxoff = sx > 3;
   lxoff = lx > 3;
   if (mask = 0xff > ( sx & 0x07 ) ) {
     if ( sxoff == lxoff )
       mask &= 0xff < 7 - (lx  & 0x07 );
     {
     register int i = pmasks_tbl -> nmasks;
     unsigned char mod_masks[ MAX_MASKS ];
     while ( -- i >= 0 )
       mod_masks[ i ] =
       pmasks_tbl -> mask_arr[ i ] & mask;
     _multiband( sxoff, pmasks_tbl -> nmasks,
       mod_masks, sy, ly );
     }
   }
   if ( lxoff > sxoff ){
   while ( ++ sxoff  lxoff )
   _multiband( sxoff, pmasks_tbl -> nmasks,
   pmasks_tbl -> mask_arr, sy, ly );
   if ( mask = 0xff <  ( 7 - ( lx  & 0x07 ) ) ) {
   register int i = pmasks_tbl -> nmasks;
   unsigned char mod_masks[ MAX_MASKS ];
   while ( -- i >= 0 )
   mod_masks[ i ] =
   pmasks_tbl -> mask_arr[ i ] & mask;
   _multiband( sxoff, pmasks_tbl -> nmasks,
     mod_masks, sy, ly );
     }
   }
 }
 _resetega();
 return(0);
}
```

map.h

```
# define MAX_MAP_NUM6
# define POSX1
# define POSY2
# define NEGX3
# define NEGY4
# define X_MAP( p, x )(float)( (p) -> usr.tlx ) +\
 (float) ABS( (x) - (p) -> usr.min_x )\
 * (p) -> x_map
# define Y_MAP( p, y )(float)( (p) -> usr.tly ) +\
 (float) ABS( (y) - (p) -> usr.max_y )\
 * (p) -> y_map
typedef struct {
 Mapping usr;
 struct physical {
   int ox, oy;
 } phy;
 float  x_map;
 float  y_map;
} Map;
typedef struct {
 int x;
 int y;
} Cartesian;
extern Map  map_tbl[];
# define VALORIGIN( p )\
{\
 if ( (p) -> ox > (p) -> max_x ||\
 (p) -> ox < (p) -> min_x ) {\
   gerror( INVOX, (p) -> ox );\
   return -1;\
 }\
 if ( (p) -> oy > (p) -> max_y ||\
 (p) -> oy < (p) -> min_y ) {\
   gerror( INVOY, (p) -> oy );\
   return -1;\
 }\
```

```
}
# define VALMAPNUM( x )\
{\
 if ( (unsigned) (x) >= MAX_MAP_NUM ) {\
   gerror( INVMPN, (x) );\
   return -1;\
 }\
}
# define VALMAPXY( map_num, ppts )\
{\
 register Mapping *p = & map_tbl[ (map_num) ].usr;\
 if ( (ppts) -> x < p -> min_x ||\
 (ppts) -> x > p -> max_x ) {\
   gerror( INVMAPX, (ppts) -> x );\
   return -1;\
 }\
 if ( (ppts) -> y < p -> min_y ||\
 (ppts) -> y > p -> max_y ) {\
   gerror( INVMAPY, (ppts) -> y );\
   return -1;\
 }\
}
# define VALPTTYPE( pt_type )\
{\
 if ( (unsigned int) pt_type > 4) {\
   gerror( INVPTTYPE, (pt_type) );\
   return -1;\
 }\
}
```

map.c

```
Map   map_tbl[ MAX_MAP_NUM ];
int
map( map_num, mpg )
intmap_num;
Mapping*mpg;
{
# ifdef DEBUG
{
# include "debug.h"
 gfunction = MAP;
 VALMAPNUM( map_num );
 VALORIGIN( mpg );
 VALXY( mpg -> tlx, mpg -> tly );
 VALXY( mpg -> brx, mpg -> bry );
}
# endif
 {
   register Map*p = &map_tbl[ map_num ];
   p -> usr = *mpg;
   p -> x_map = (float) ( mpg -> brx - mpg -> tlx ) /
     (float) ABS( mpg -> max_x - mpg -> min_x );
   p -> y_map = (float) ( mpg -> bry - mpg -> tly ) /
     (float) ABS( mpg -> max_y - mpg -> min_y );
   p -> phy.ox = X_MAP( p, mpg -> ox );
   p -> phy.oy = Y_MAP( p, mpg -> oy );
 }
 return 0;
}
```

H.2 SOAP Listing

PC-end

soap.h

```
# ifndef _SOAP
# define _SOAP
# define SIMPLEX(01)
# define DUPLEX(02)
# define NPORTS(02)
# define COM1_PORT(00)
# define COM2_PORT(01)
# define PROC_INTR 0
# define IGNR_INTR 1
# define KB__HIT-1
# define BAD_FIL-2
# define TIM_OUT-3
# define GEN_ERR-4
# define NOT_INI-5
# define NOT_MOD-6
# define ERR_REQ-7
# define SND_FAIL-8
# endif
```

cntl.asm

```
ISRENT_TEXTSEGMENT  BYTE PUBLIC 'CODE'
ISRENT_TEXTENDS
_DATA  SEGMENT  WORD PUBLIC 'DATA'
_DATA  ENDS
CONST SEGMENT  WORD PUBLIC 'CONST'
CONST ENDS
_BSS SEGMENT  WORD PUBLIC 'BSS'
_BSS ENDS
DGROUP  GROUPCONST,_BSS,_DATA
 ASSUME
CS:ISRENT_TEXT,DS:DGROUP,SS:DGROUP,ES:DGROUP
ISRENT_TEXTSEGMENT
 PUBLIC   __disint
__disint PROC FAR
 cli
 retf
__disint ENDP
 PUBLIC   __enbint
__enbint PROC FAR
 sti
 retf
__enbintENDP
ISRENT_TEXTENDS
END
```

isrent.asm

```
ISRENT_TEXTSEGMENT  BYTE PUBLIC 'CODE'
```

```
ISRENT_TEXTENDS
_DATA SEGMENT WORD PUBLIC 'DATA'
_DATA ENDS
CONST SEGMENT WORD PUBLIC 'CONST'
CONST ENDS
_BSS SEGMENT WORD PUBLIC 'BSS'
_BSS ENDS
DGROUP GROUPCONST,_BSS,_DATA
ASSUME
CS:ISRENT_TEXT,DS:DGROUP,SS:DGROUP,ES:DGROUP
_BSS SEGMENT
stktop DW512 DUP(?)
stkbtm DW  1 DUP(?)
oldsp DW 1 DUP(?)
oldss DW 1 DUP(?)
_BSS ENDS
EXTRN __rxisr:FAR
EXTRN __txisr:FAR
ISRENT_TEXTSEGMENT
 PUBLIC __com1isr
__com1isr PROC FAR
 push ax
 push ds
 mov   ax,DGROUP
 mov   ds,ax
 mov   oldsp,sp
 mov   oldss,ss
 mov   sp,OFFSET DGROUP: stkbtm
 mov   ss,ax
 push bx
 push cx
 push dx
 push es
 push di
 push si
 xor  ax,ax
 push ax
 mov   dx,1018
 in  al,dx
 and  al,6
 cmp   al,4
 jne  $I12
 call FAR PTR __rxisr
 jmp  SHORT $I13
$I12:
 call FAR PTR __txisr
$I13:
 add  sp,2
 popsi
 popdi
 pop es
 popdx
 popcx
 popbx
 mov  sp,oldsp
 mov  ss,oldss
 popds
 mov  al,20h
 out 20h,al
 popax
 iret
__com1isr ENDP
 PUBLIC __com2isr
__com2isr PROC FAR
 push ax
 push ds
 mov   ax,DGROUP
 mov   ds,ax
 mov   oldsp,sp
 mov   oldss,ss
 mov   sp,OFFSET DGROUP: stkbtm
 mov   ss,ax
 push bx
 push cx
 push dx
 push es
 push di
 push si
 mov   ax,1
 push ax
 mov   dx,762
 in  al,dx
 and  al,6
 cmp   al,4
 jne  $I15
 call FAR PTR __rxisr
 jmp  SHORT $I16
$I15:
 call FAR PTR __txisr
$I16:
 add  sp,2
 popsi
 popdi
 pop es
 popdx
 popcx
 popbx
 mov  sp,oldsp
 mov  ss,oldss
 popds
 mov  al,20h
 out 20h,al
 popax
 iret
__com2isr ENDP
ISRENT_TEXTENDS
END
```

soapini.c

```
Ctlport_ctlbuf[ NPORTS ] = {
 {
   _com1isr, 0x3f8, STATE_IDLE,
   {
     0, 0, ACC_PKT, 0, RX_BUFFER(0),
     RX_BUFFER(0), RX_BUFFER(0),
     0, 0, 0, 0
   },
   {  0, 0,  TX_BUFFER(0),
     TX_BUFFER(0)
   }
 },
 {
   _com2isr, 0x2f8, STATE_IDLE,
   {
     0, 0, ACC_PKT, 0, RX_BUFFER(1),
     RX_BUFFER(1), RX_BUFFER(1),
     0, 0, 0, 0 },
   { 0,  0, TX_BUFFER(1),
     TX_BUFFER(1)
   }
 }
};
void
soap_init( port )
int port;
{
 _ctlbuf[ port ].tx.tx_seq_no = 0;
```

```
 _ctlbuf[ port ].rx.rx_seq_no = 0;
}
void
_port_init( port_id, baud_rate )
int port_id;
int baud_rate;
{
 register Ctlport *p_ctlbuf = & _ctlbuf[ port_id ];
 p_ctlbuf -> port_base = _base_reg( port_id );
 p_ctlbuf -> rx.count = 0;
 p_ctlbuf -> rx.p_avail = 0;
 p_ctlbuf -> rx.p_len = 0;
 p_ctlbuf -> rx.rx_seq_no = 0;
 p_ctlbuf -> state = STATE_IDLE;
 p_ctlbuf -> rx.wptr = p_ctlbuf -> rx.rptr
   = RX_BUFFER( port_id );
 _config( port_id, baud_rate, p_ctlbuf -> isr );
}
void
_flush_rx_buf( port_id )
int port_id;
{
 register Ctlport *p_ctlbuf = & _ctlbuf[ port_id ];
 _disint();
 p_ctlbuf -> rx.count = 0;
 p_ctlbuf -> rx.p_avail = 0;
 p_ctlbuf -> rx.p_len = 0;
 p_ctlbuf -> rx.p_sts = IGN_PKT;
 p_ctlbuf -> rx.wptr = p_ctlbuf -> rx.rptr
   = RX_BUFFER( port_id );
 _enbint();
}
void
_set_state( port_id, state )
int port_id;
int state;
{
 DBGMSG3( "SET_STATE:: %d %d", port_id, state );
 _ctlbuf[ port_id ].state = state;
}
```

soapsend.c

```
static Packet  *form_pkt();
# define DMSG1 "SOAP_SEND: packet sent, seq_num = %d"
# define DMSG2 "SOAP_SEND: Recvd %x, %x %x"
# define DMSG3 "SOAP_SEND:: port = %d, data_size = %d"
int
soap_send( port, data, data_size )
int port;
unsigned char *data;
int data_size;
{
 long time();
 long start_time;
 register int retry_count = 3;
 register int trans_seq = _ctlbuf[ port ].tx.tx_seq_no;
 register Packet *p =
   form_pkt( data, data_size, trans_seq );
 DBGMSG3( DMSG3, port, data_size );
 time( & start_time );
 for (;  ;  ) {
   _set_state( port, STATE_SEND );
   _flush_rx_buf( port );
   if ( _write_port( port,(unsigned char *)p,
     PKT_SIZE(p)) == -1){
     _set_state( port, STATE_IDLE );
     return -1;
   }
  DBGMSG2( DMSG1, trans_seq );
  for ( ; ; ){
   Packet    pkt;
   if ( time(0L) - start_time > PC_ACK_TIM_OUT ){
     if (  retry_count > 0 ){
       -- retry_count;
       break;
     }
     _set_state( port, STATE_IDLE );
     return -1;
   }
   if ( _read_port( port, (unsigned char *)&pkt ) == 0 )
     continue;
   DBGMSG4( DMSG2, pkt.cntl_byte, pkt.seq_num,
     pkt.data_size );
   switch ( pkt.cntl_byte  ){
     case  ACK:
       if ( pkt.seq_num == trans_seq ){
           _ctlbuf[ port ].tx.tx_seq_no =
             NXT_SEQ_NO( trans_seq );
         return 0;
       }
       _set_state( port, STATE_SEND );
     case  SOH:
       continue;
     case  NAK:
       if ( -- retry_count < 0 )
         return -1;
   }
   break;
  }
 }
}
static Packet *
form_pkt( data, n, trans_seq )
unsigned char *data;
register int  n;
unsigned char trans_seq;
{
 static char   buf [ MAX_DPKT_SIZE ];
 register Packet *phdr = ( Packet * )buf ;
 register unsigned char *q = data;
 register unsigned char check_sum = n;
 phdr -> data_size = n;
 {
   register unsigned char *p = DATA( phdr );
   do  {
     check_sum ^= ( *p ++ = *q ++ );
   } while ( -- n > 0 );
   *p = check_sum ^ ( phdr -> seq_num = trans_seq ) ;
 }
 phdr -> cntl_byte = SOH;
 return phdr;
}
```

soaprecv.c

```
int
soap_recv( port_id, buf, time_out )
int port_id;
unsigned char *buf;
int time_out;
{
 long start_time, time() ;
 time ( & start_time );
```

```
do {
  static Packetpkt;
  if ( _read_port( port_id,
    (unsigned char *) &pkt ) == 0 )
    continue;
  if ( pkt.cntl_byte == SOH ){
    register int info_size = INFO_SIZE( &pkt );
    register unsigned char*p_src = DATA( &pkt );
    register unsigned char*p_dest = buf;
    register int n;
    for( n = info_size ; n > 0 ; -- n )
      *p_dest ++ = *p_src ++;
    return info_size;
  }
} while ( time( 0L ) - start_time <= time_out );
return -1;
}
```

soapint.h

```
# ifndef _SOAPINT
# define _SOAPINT
# include "soapism.h"
# define MAX_INFO_LEN 128
# define MAX_DATA_LEN ( MAX_INFO_LEN + sizeof( Creqres ) )
# define MAX_SEQ_NO127
# define MIN_PKT_SIZE ( sizeof( Ack ) )
# define PKT_INF_SIZE ( sizeof( ( (Packet *)0) -> info ) )
# define PKT_HDR_SIZE ( sizeof( Packet ) - PKT_INF_SIZE )
# define MIN_DPKT_SIZE ( PKT_HDR_SIZE + 1 )
# define MAX_DPKT_SIZE ( sizeof( Packet ) )
# define SEQ_NUM(p) ( ((Packet *)(p)) -> seq_num )
# define ACK_SEQ_NUM(p) ( ((Ack *)(p)) -> seq_num )
# define NXT_SEQ_NO( n ) ( ( (n)  == 127 ) ? 1 : (n) + 1 )
# define DATA( p ) ( &((Packet *)(p)) -> info[0] )
# define DATA_LEN( p ) ( ((Creqres *)(p)) -> msg_length )
# define INFO_SIZE( p ) ( ((Packet *)(p)) -> data_size )
# define MIN( x, y ) ( (x) <= (y) ? (x) : (y) )
# define PKT_SIZE( p ) ( ((Packet *)(p)) -> data_size\
 + MIN_DPKT_SIZE )
# define MSG_SIZE( p ) ( PKT_SIZE( p ) - 1 )
# define SOH ((unsigned char)0x81)
# define ACK ((unsigned char)0x86)
# define NAK ((unsigned char)0x95)
# define SNOERR 0x01
# define CSMERR 0x02
# define STXERR 0x03
# define PTYERR 0x11
# define FRMERR 0x12
# define OVNERR 0x13
# define DESERR 0x21
# define NOBERR 0x22
# define ISTERR 0x31
# define MULERR 0x40
# define MISERR 0x41
# define SUCCESS 0x00
# define CMDREQ 0x01
# define CMDRES 0x02
# define CMDRET 0x03
# define FETCH 0x0
# define DATE 0x1
# define TIME 0x2
# define OPEN 0x3
# define CLOSE 0x4
# define READ 0x5
# define SEEK 0x6
# define WRITE 0x7
# define UNLINK 0x8
# define ISFETCH 0x9
# define ISSTAT 0xa
typedef struct Creqres {
 unsigned charmsg_length;
 unsigned charmsg_type;
 unsigned charcommand;
 unsigned charseq_num;
 long       status;
} Creqres;
typedef struct Fetch {
 long       offset;
 long       numbyt;
} Fetch;
typedef struct {
 unsigned char acc_mode;
 unsigned char max_rec;
 unsigned short reclen;
 Is_keydesc  lo_key;
 Is_keydesc  hi_key;
 long      mtime;
} Isfetch;
# define ERR_OPEN(-1)
# define ERR_READ(-2)
# define ERR_SEEK(-3)
# define ERR_STAT(-4)
# define NOT_MODI(-5)
typedef struct Packet {
 unsigned charcntl_byte;
 unsigned charseq_num;
 unsigned chardata_size;
 char  info[ MAX_DATA_LEN + 1];
} Packet;
typedef  struct Nak{
 unsigned charcntl_byte;
 unsigned charseq_num;
 unsigned charcode;
 char       null;
} Nak;
typedef  struct Ack{
 unsigned charcntl_byte;
 unsigned charseq_num;
 char    null;
 char    filler;
} Ack;
# endif
```

IOP-end

main.c

```
intchild_pid = -1;
char *procname;
static char  *baud_string;
static char  *device_name;
int
main( argc, argv )
int argc;
char *argv[];
{
 register int  baud_code;
 int pfd[ 2 ];
 signal( SIGINT, SIG_IGN ) ;
 signal( SIGQUIT, SIG_DFL );
 procname = argv[ 0 ];
```

```
if( argc < 2 || argc > 3 )
  fatal( "Usage: %s line [ baud ]", procname );
{
  register int i = 3;
  do {
    close(i);
  } while ( ++ i < 20 );
}
( void )open_device( device_name = argv[ 1 ] );
baud_string = ( argc == 3 ) ? argv[2] : DEFAULT_BAUD;
if ( (baud_code = get_baud ( atol(baud_string) )) == -1 )
  fatal( "illegal baud rate" );
( void )init_device( baud_code );
if ( pipe ( pfd ) == -1 )
  fatal( "pipe failure" );
DBGMSG3( "Created pipe rd(%d) wr(%d)", pfd[0], pfd[1] );
CONNECT( pfd[ 0 ], R_PIPE_FD );
if ( (child_pid = fork() ) == 0 ){
  char wfd[ 10 ];
  sprintf( wfd, "%d", pfd[1] );
  execlp( READER_PROCESS, READER_ARG0, wfd, 0 );
  fatal( "cannot exec %s", READER_PROCESS );
} else if ( child_pid == -1 ){
  fatal( "cannot fork" );
}
signal( SIGCLD, mourn_chlds_death );
close( pfd[ 1 ] );
( void )process_messages();
}
int
get_baud( baud_rate )
int baud_rate;
{
static struct baud_map {
  int baud_rate;
  int code;
} baud_map[ ] ={
  { 50, B50 },
  { 75, B75 },
  { 110, B110 },
  { 134, B134 },
  { 150, B150 },
  { 200, B200 },
  { 300, B300 },
  { 600, B600 },
  { 1200, B1200 },
  { 1800, B1800 },
  { 2400, B2400 },
  { 4800, B4800 },
  { 9600, B9600 }
};
register int n =
sizeof( baud_map ) / sizeof( struct baud_map );
register struct baud_map *p = baud_map;
do {
  if ( p -> baud_rate == baud_rate )
    return p -> code;
  p ++ ;
} while ( -- n > 0 );
return -1;
}
void
mourn_chlds_death()
{
static char msg[] =
  "%s: killing self due to death of child\n";
DBGMSG2( msg, procname );
fprintf( stderr, msg, procname );
exit(1);
}
```

cri.c

```
extern char *strncpy(), *strchr(), *getenv();
extern char *resol_fl();
# define TIME_ST_LEN16
# define FNAM_ST_LEN( MAX_INFO_LEN - sizeof( Fetch ) + 1 )
# define TABLE_SIZE( tbl, type ) \
 ( sizeof( tbl )/sizeof( struct type ) )
void
cmd_interpret( creqres )
Creqres *creqres;
{
  static struct perform{
  void (* p_func) ();
} perform_tbl[] = {
  p_fetch,
  p_datetime,
  p_datetime,
  p_open,
  p_close,
  p_read,
  p_lseek,
  p_write,
  p_unlink,
  p_isfetch,
  p_isstat,
  p_kill
};
if ( creqres -> command >= 0 && creqres -> command <
sizeof( perform_tbl ) / sizeof( struct perform ) )
(* perform_tbl[ creqres -> command ].p_func) ( creqres );
return;
}
p_datetime( creqres )
Creqres *creqres;
{
char data[ MAX_DATA_LEN ];
register Creqres *p = (Creqres *) data;
DBGMSG1( DMSG8 );
if ( time( (long *)( p + 1 ) ) == -1 ){
  DBGMSG1( DMSG5 );
  p -> status = ERR_TIME;
  p -> msg_length = sizeof( Creqres );
}
else {
  p -> msg_length = sizeof( Creqres ) + sizeof( long );
  p -> status = sizeof( long );
  DBGMSG2( DMSG10, (long *)( p + 1) );
}
p -> seq_num = 1;
p -> msg_type = CMDRES;
p -> command = creqres -> command;
snd_data( p );
}
```

H.3 GIM Listing

main.h

```
# ifndef _MENU
# define _MENU
```

```
# define MPAGE2
# define NPAGE5
# define TSLICE30
# define BAUD9600
# define MENU'M'
# define SELECT'S'
# ifdef NEXT
# undef  NEXT
# endif
# define NEXT'N'
# define LOCK'L'
# define UNLOCK'U'
# define QUIT 'Q'
# define EXPAND'E'
# define PARENT 'P'
# define DUMP'D'
# define REPLAY'R'
# define CAN_ALM 'C'
# define PAGE_SYS0
# define PAGE_PERAM1
# define PAGE_IOP -1
# define PAGE_CM 2
# define PAGE_BM3
# define PAGE_CARD4
# define NEXT_PAGE( page_no ) {\
 if ( ++ page_no >= MPAGE )\
   page_no = 0;\
}
# define MENU_PRINT( item_no, color )\
 prowfrm( XMAX > 1, ( ITEM_ROW + (item_no < 1)) * HCHAR,\
   color, page_func[ item_no ] . name, FRM_CEN, 0 );
typedef struct {
 char *name;
 int  bg_color;
 int  bg_style;
 int  parent_page;
 int (far * getinfo  )();
 void (far * drawframe )();
 void (far * putvalues )();
 int  (far * select )();
 int (far * nextobj )();
} PageFunc;
extern PageFuncpage_func[];
# define F_COLORLI_WHITE
# define B_COLORLI_BROWN
# define MENU_ROW5
# define ITEM_ROW15
extern int exist_page, next_page;
# endif
```

main.c

```
static void show_page(),gim_setup(),menu();
static int analyse_kb();
static int change_screen = 1;
int exist_page = 0;
int next_page = 0;
int soap_config = SIMPLEX;
static  char see_above[] = "See Above";
# define DISP_SEE_ABOVE() error_log( see_above )
PageFunc page_func[] = {
{
  " SYSTEM STATUS ",
  LI_BLUE, FILL_EXPLD, -1,
  ss_getfiles, ss_frame, ss_putvalues,
  ss_select, ss_next
},
{
  " EXCHANGE PERFORMANCE ",
  LI_BLUE, FILL_EXPLD, -1,
  ep_getfiles, ep_frame, ep_putvalues,
  ep_select, ep_next
},
{
  " CENTRAL MODULE STATUS ",
  LI_CYAN, FILL_CRUSH, PAGE_SYS,
  cm_getfiles, cm_frame, cm_putvalues,
  cm_select, cm_next
},
{
  " BASE MODULE STATUS ",
  LI_CYAN, FILL_EXPLD, PAGE_SYS,
  bm_getfiles, bm_frame, bm_putvalues,
  bm_select, bm_next
},
{
  " CARD STATUS ",
  LI_BROWN, FILL_EXPLD, 0,
  cd_getfiles, cd_frame, cd_putvalues,
  cd_select, cd_next
},
{
  " BASE MODULE PERFORMANCE ",
  LI_CYAN, FILL_CRUSH, PAGE_PERAM,
  bp_getfiles, bp_frame, bp_putvalues,
  bp_select, bp_next
}
};
main( argc, argv )
int  argc;
char *argv[];
{
 long start_time;
 long time();
 if ( argc > 2 )  {
   fprintf( stderr,
 "Usage : GIM [ Duplex : 1, Simplex : default ]\n" );
   exit(1);
 }
 if ( argc == 2 ){
   if ( atoi( argv[1] ) == 1 )
     soap_config = DUPLEX;
 }
 gim_setup();
 error_log( "Hit any key to continue" );
 getch();
 while ( 1 ) {
   show_page();
   time( & start_time );
   while ( 1 ) {
     showclock( page_func[ exist_page ].bg_color
     | HI_INTENSITY );
     if ( kbhit() )
       if ( analyse_kb()) {
         change_screen = 1;
         break;
       }
     if ( (time(0L) - start_time) >= TSLICE ) {
       if ( ! islock() )  {
         int parent =
         page_func[ exist_page ].
         parent_page;
         if ( parent < 0 ) {
         NEXT_PAGE( next_page );
         if(next_page!=exist_page)
         change_screen = 1;
         break;
```

H.4 Snapshots