

716

AUDIT PROCESS IN C-DOT DIGITAL SWITCHING SYSTEM

**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

SUNIL SINDWANI

School of Computer and System Sciences

Jawaharlal Nehru University

New Delhi


April, 1989

994

Certificate

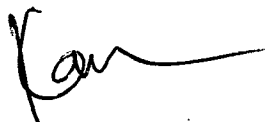
This work titled : **AUDIT PROCESS IN C-DOT DIGITAL SWITCHING SYSTEM** has been carried out by Mr. Sunil Sindwani, a bonafide student of School of Computer and System Sciences, Jawaharlal Nehru University.

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.



SUNIL SINDWANI

Student



Prof. Karmeshu

Supervisor

SCSS, JNU.



Prof. B. S. Khurana

Supervisor

SCSS, JNU.



Prof. Karmeshu

Dean,

SCSS, JNU.

Abstract

Fault conditions, can be disastrous for any system, especially for systems which are required to work continuously. A public switching system falls under this category. It forms the infrastructure required for communication systems and a fault in it can not be afforded. By introducing redundancy, systems can be designed, which can detect and recover from fault conditions. In such systems, duplicate copies of hardware is kept. One of them is always working. The other one is kept as stand-by. In the event of detection of fault in active unit, a smooth switch-over is performed from active to stand-by unit without disrupting any ongoing process. The switch-over is performed by recovery sub- system. The duplication, however, involves another vital factor - Availability of up-to-date data to stand-by unit, to be able to take over from active unit. Further, such a mechanism is also required to ensure consistency of global data present on different processors in a distributed processing environment.

The task of providing up-to-date data to stand-by unit is a complicated one. In an environment which supports distributed processing, in addition to fault tolerance, it becomes even more complicated. An obvious way of accomplishing this task can be - To update the data in stand-by unit, every time it is updated in active unit. However, there should be some kind of checking mechanism by which consistency in both sets of data can be ensured. Precisely, this is the task of AUDITS - To ensure the consistency of data by performing consistency checks (audits), routinely and on request on different sets of duplicated data.

Several factors affect the design of audits. Synchronization problems and also race

conditions, surface as different processes try to access same set of global data. Further, holes in data area (unused bytes) can lead to unmeaningful auditing. The audit process provides software support to distributed and fault tolerant computing environment but at the same time it loads the system. A suitable compromise between extent of support and load generated should also be worked out.

This thesis describes various aspects of, design and implementation of audits in C-DOT Digital Switching System. The work was carried out at C-DOT and successfully implemented.

Contents

Certificate	i
Abstract	iii
Preface	viii

Chapter 1 **1**

C-DOT Digital Switching System	1
1.1 Hardware Architecture	1
1.1.1 Basic Modules - 1	
1.1.1.1 Base Module - 2	
1.1.1.2 Administrative Module - 2	
1.1.1.3 Central Module - 2	
1.1.1.4 Input Output Processor - 3	
1.1.2 Switching System Configurations - 4	
1.1.2.1 Single Base Module (SBM) - 4	
1.1.2.2 Multi Base Module (MBM) - 4	
1.2 Software Architecture	4
1.2.1 Overview - 4	
1.2.2 Sub Systems Integration - 5	
1.2.2.1 Inter Process Communication - 6	
1.3 Development Environment	6

1.3.1 Why UNIX and 'C' - 7

Chapter 2 **8**

Fault Tolerance 8

 2.1 Fault Tolerance in C-DOT DSS 9

 2.1.1 Duplex IOP Architecture - 9

 2.1.2 Maintenance Software On IOP - 9

 2.2 Data Organization 10

 2.2.1 File Management Systems - 10

 2.2.1.1 UNIX files - 10

 2.2.1.2 C-ISAM files - 11

 2.3 Processes On IOP 11

Chapter 3 **12**

Audit Process 12

 3.1 Audit Requirements 12

 3.1.1 IOP - IOP (Disc to disc) audits - 12

 3.1.2 IOP - BP (Disk to memory) audits - 12

 3.2 Invocation 13

 3.2.1 System Initiated Auditing - 13

 3.2.2 Calendar Based Routine Auditing - 13

 3.2.3 Idle Time Audits - 13

 3.2.4 Operator Initiated Audits - 13

 3.3 Design Considerations 14

 3.4 Auditing Strategy 15

- 3.4.1 IOP - IOP Audits - 15
 - 3.4.1.1 File Level - 15
 - 3.4.1.2 Record Level - 16
- 3.4.2 IOP - BP Audits - 17
 - 3.4.2.1 Single Base Module - 17
 - 3.4.2.2 Multi Base Module - 17

Chapter 4 **18**

Implementation - I 18

- 4.1 IOP - IOP Audits 18
 - 4.1.1 Explanation of algorithm - 24
- 4.2 IOP - BP Audits 26
 - 4.2.1 Explanation - 27

Chapter 5 **29**

Implementation - II 29

- 5.1 IOP-IOP Audits 30
- 5.2 IOP-BP Audits 36

Chapter 6 **37**

Conclusion 37

- 6.1 Correctness of master file : 37
- 6.2 Deletion of record : 38
- 6.3 Locking : 38
- 6.4 Memory locks : 38
- 6.5 C-ISAM 39

6.6 CRC 39

6.7 Dummy bytes in C-DOT data structures : 40

 6.7.1 System generated dummy bytes - 40

 6.7.2 User generated dummy bytes : - 41

6.8 Further Scope 41

Appendix A **A-One**

Abbreviations And Glossary A-One

Appendix B **B-One**

System And Function Calls B-One

 B.1 UNIX System Calls B-One

 B.2 'C' Function Calls B-Three

 B.3 HDLC Library Function Calls B-Five

 B.4 C-ISAM Library Function Calls B-Five

Appendix C **C-One**

Source Code C-One

 C.1 The Organization C-One

 C.2 The Code C-One

Appendix D **D-One**

References and Bibliography D-One

Preface

My urge to work in a real time environment led me to the Centre for Development of Telematics (C-DOT), India's Telecom Technology Centre. C-DOT is currently engaged in the development of a Digital Switching System (DSS).

Earlier, I along with my classmate Arun Viswanathan, had discussed the possibility of undertaking project work at C-DOT with Prof. Karmeshu, Dean, School of Computer and System Sciences, and Prof. B. S. Khurana, School of Computer and System Sciences, Jawaharlal Nehru University. Their encouraging response led us to apply at C-DOT.

I was entrusted with the development of an AUDIT PROCESS which is the topic of my dissertation. This project was carried out under the guidance of Mr. Lalit Dhingra, Group Leader - UNIX Group.

Audits

The typical requirement of most of the Digital Switching Systems and other real time applications is FAULT TOLERANCE. On systems which are required to work continuously, a fault condition can be disastrous. To ensure an error free system, one possibility is duplication of Hardware. By introducing a measure of redundancy a system capable of fault-detection and correction can be built.

Redundancy at Hardware level requires duplication of processors. At any given instant, one of them is in ACTIVE state and the other one in STAND-BY.state. In case of a

fault, a switch over is performed from ACTIVE to STAND-BY system. The fault tolerant software sub-system accomplishes the switch over task without disrupting any part of the process. However, in case of software, duplication of data is also required. One set for active module and other one for stand-by module. A crucial factor in design of fault tolerant systems is availability of up-to-date data to stand-by unit. To achieve this, periodic updation and auditing of data is performed. Audits are responsible for performing a consistency check on different sets of duplicated data. In case of inconsistencies, data is updated from active system. This thesis concentrates on AUDIT subsystem of C-DOT DSS.

Organization of Thesis

This thesis has been organized in six chapters and four appendices. Bibliography and references have been included at the end. For conciseness, only such information which is considered absolutely necessary is provided.

Since, while working in a team, a designer has to stick to a specified frame, it is desirable to familiarize the reader with the environment and specifications as given by C-DOT. Chapter I gives an overview of architecture and programming conventions adopted at C-DOT. An account of why "C" and UNIX are used for development is also provided.

Chapter 2 highlights the importance and need of performing audits after describing the organization of data in DSS.

Chapter 3 describes the audits at conceptual level and points which need a consideration at the time of designing, in connection with C-DOT architecture and requirements. It also discusses various means by which Audits can be initiated.

The implementation of main routine is discussed in Chapter 4. It also presents the algorithm. A brief description of procedure for validating Audits is also provided.

Chapter 5 explains the different functions developed in order to implement the algorithm. Different strategies used for implementing and optimizing them are also discussed, wherever necessary.

The last Chapter concludes the discussion by mentioning some problems and suggests their possible solution. Scope for further optimization at conceptual level is also discussed.

An attempt has been made to include all highly relevant but specific information in this thesis. Extensive use of abbreviations makes it necessary to provide a list of abbreviations. Appendix A provides such a list along with a glossary.

A brief description of different system and library function calls is given in Appendix B.

Appendix C contains C-DOT and audit process Hash Definitions and Type Definitions and entire source code.

Bibliography and references have been listed in appendix D.

Acknowledgments

I am grateful to Prof. Karmeshu, Dean, School of Computer and System Sciences, and Prof. B. S. Khurana, School of Computer and System Sciences, without whose valuable and constant encouragement, supervision, vital support and constructive criticism throughout the planning and completion of this work, this project would not have materialized.

I offer my sincere thanks to Mr. Lalit Dhingra, Group Leader UNIX Group C-DOT, Vinay Deo, Adesh Gupta, Mukul Goyal, all of UNIX group for their constant support and

guidance. Thanks are also due to Alka Anand, who was also involved in the design of audits and who made me stick to project deadlines, and to R. Vasudha whose friendly talks made me feel at home during the entire period of the project. I extend my thanks to R. Ramanathan and Joydeep Bose also.

I feel greatly indebted to my classmates Arun Viswanathan and Naveen Jain for their help and suggestions during the preparation of the final copy of this thesis.

Thanks are also due to C-DOT for placing its resources and facilities at my disposal, which were very essential for this project.



SUNIL SINDWANI

Chapter 1

C-DOT Digital Switching System

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. 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.

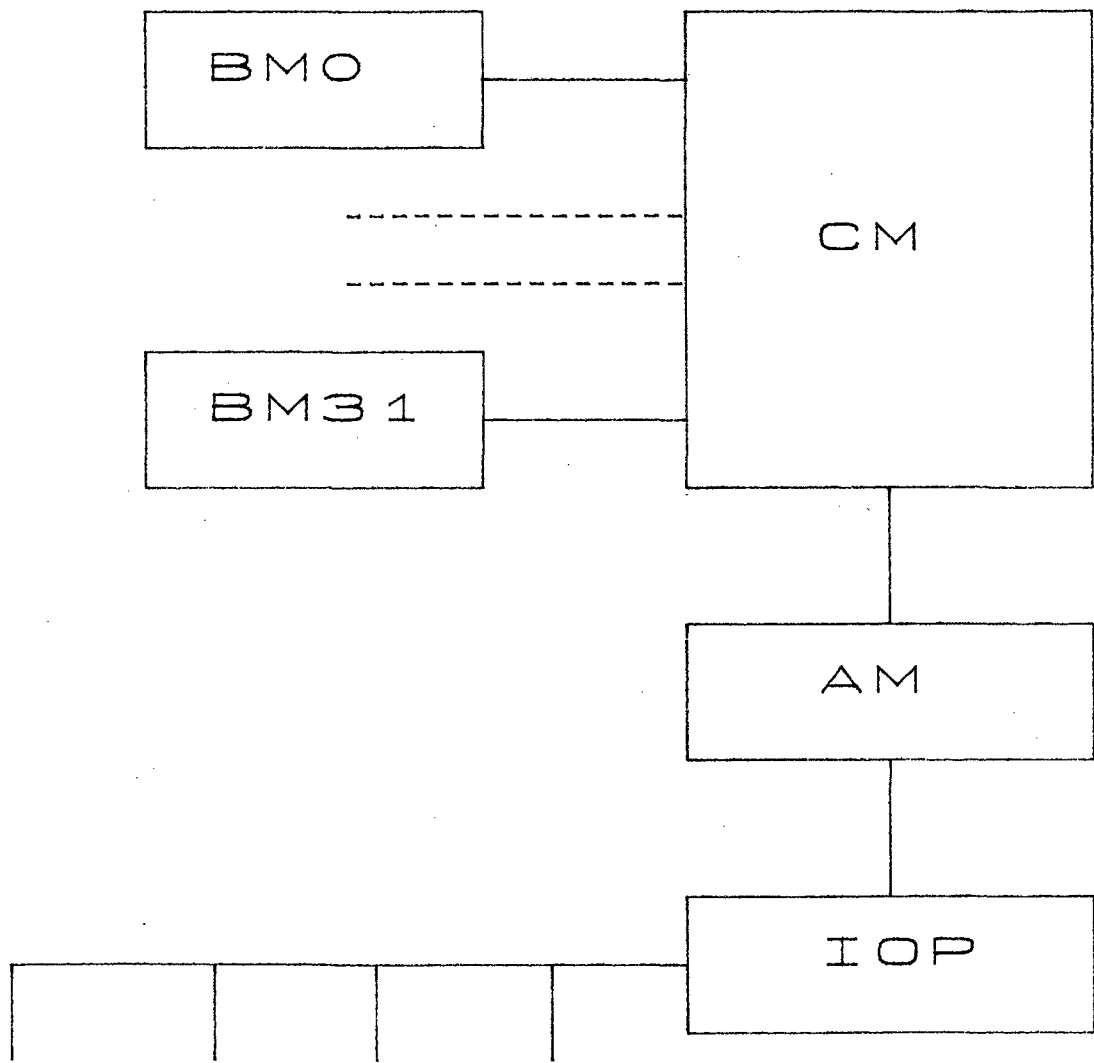
This chapter concentrates on Hardware and software organization of C-DOT DSS.

1.1 Hardware Architecture

1.1.1 Basic Modules

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

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



DISKTAPE VDU PRINTER

FIG 1.1 A TYPICAL CONFIGURATION
OF C-DOT DSS

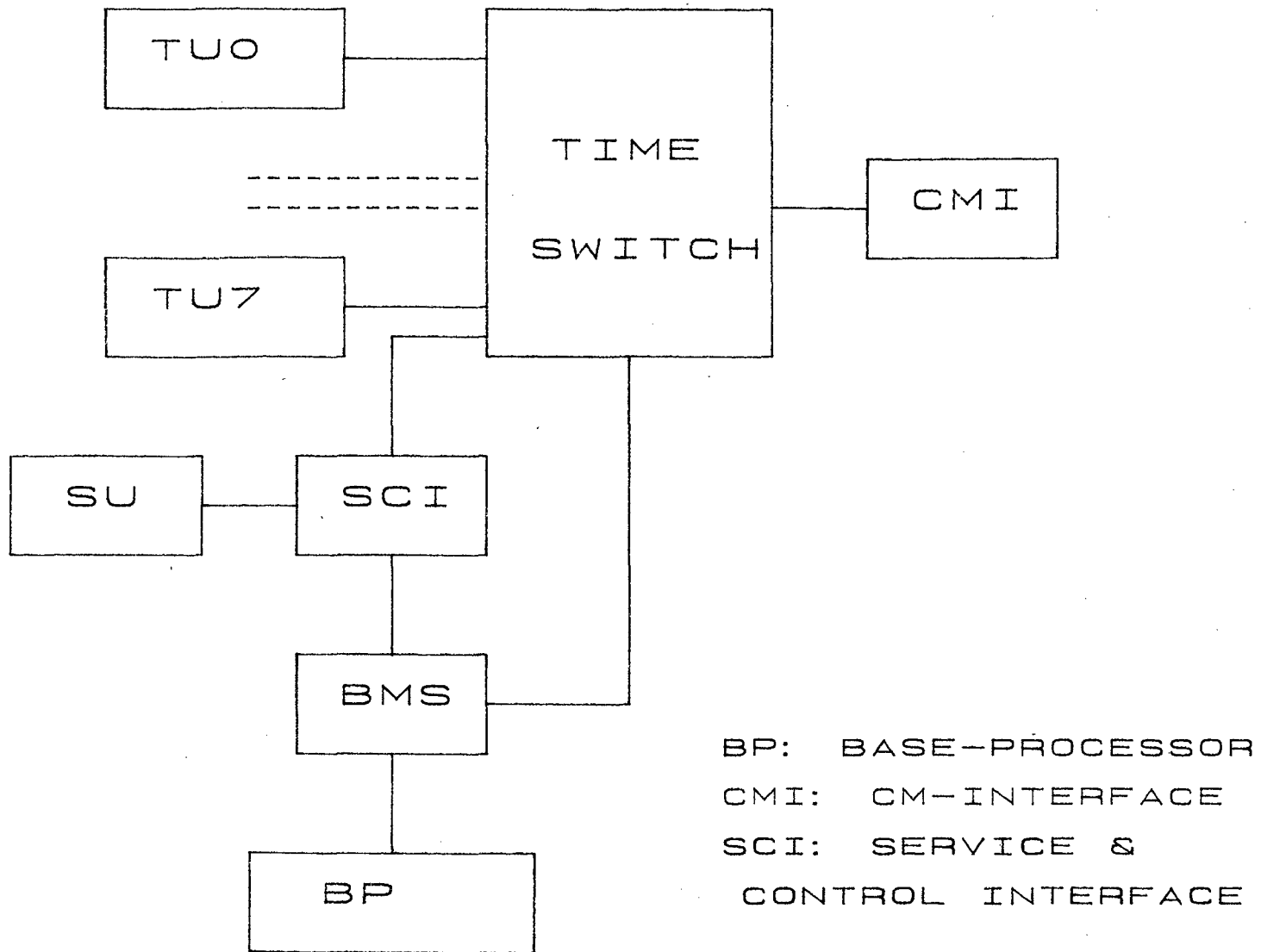


FIG 1.2 TYPICAL BM CONFIGURATION

A typical configuration of these units is shown in fig 1.1.

Although IOP is a part of AM it is discussed separately, further details are provided in C-DOT DSS ARCHITECTURE [1] and [9].

1.1.1.1 Base Module

BM is primary growth unit of C-DOT DSS. BMs may differ in the types and quantities 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 fig 1.2. 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, 68010 has a large database for controlling its working. BM is connected to other links via high speed high level data links (HDLC). A detailed description of BM can be seen in "C-DOT DSS HARDWARE ARCHITECTURE" [2].

1.1.1.2 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 basis, and passes it to IOP.

1.1.1.3 Central Module

This provides the interconnection facility for BMs. It 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 high level data links (HDLC). 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.

1.1.1.4 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.
- Fault detection and recovery
- System status display

The bus architecture is similar in characteristics to VME bus.

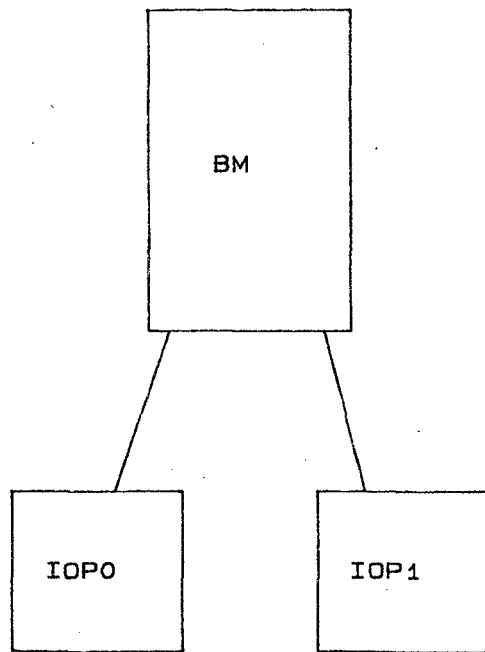


FIG 1.3a SINGLE BASE MODULE
CONFIGURATION
(DUPLEX IOPs)

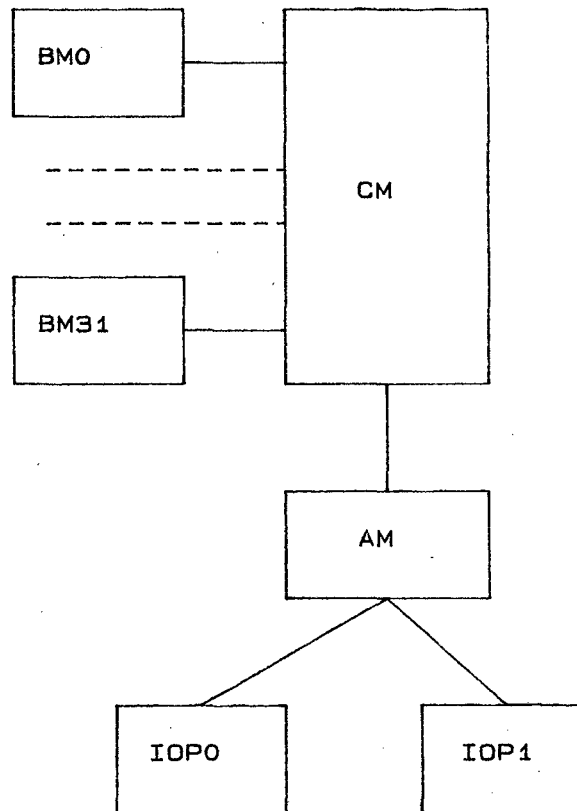


FIG 1.3b MULTI BASE MODULE
CONFIGURATION
(DUPLEX IOPs)

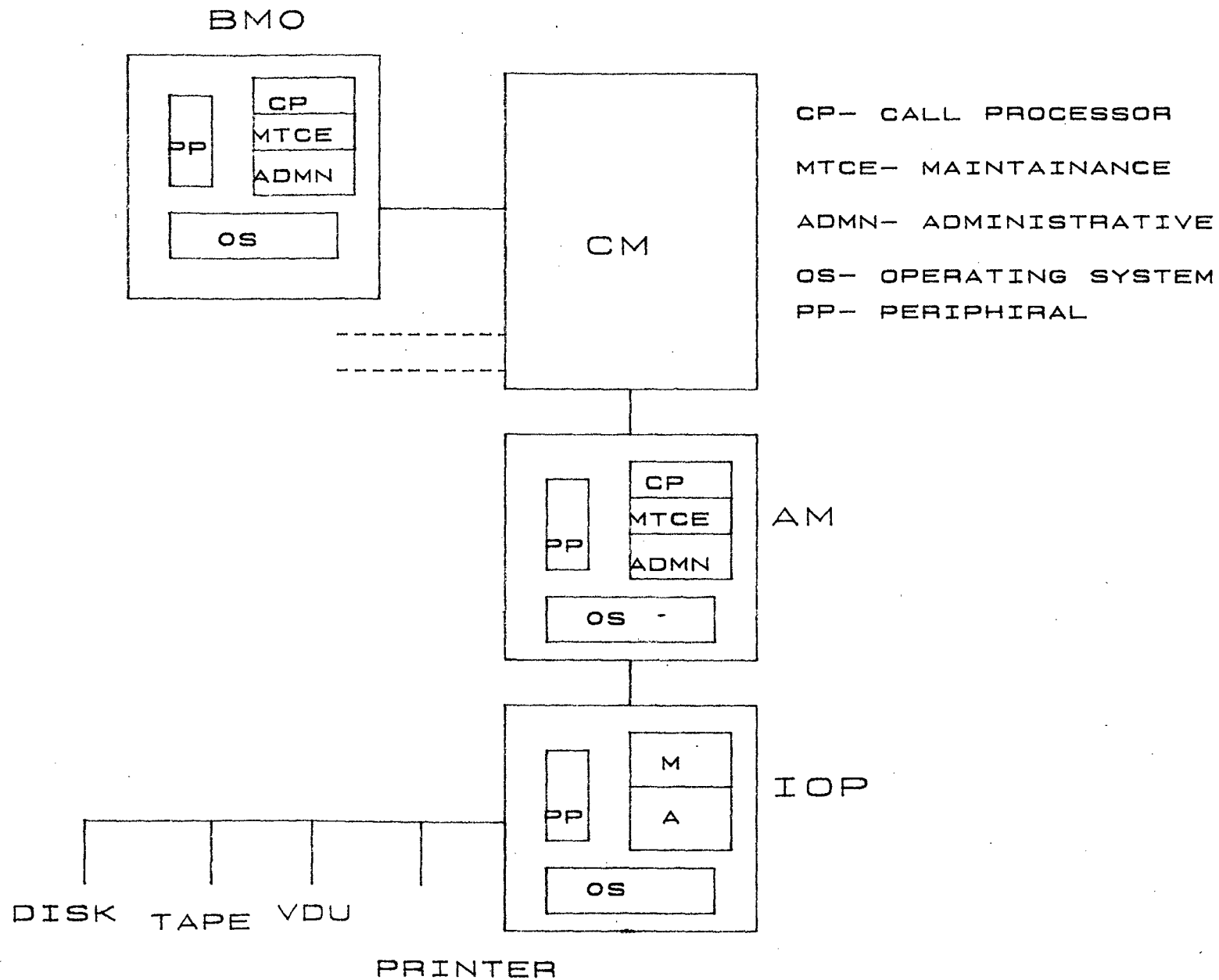


FIG 1.4 SOFTWARE ORGANIZATION

1.1.2 Switching System Configurations

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

1.1.2.1 Single Base Module (SBM)

This is smallest possible configuration using above mentioned modules. 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. Fig. 1.3a shows a single module configuration.

1.1.2.2 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 Fig. 1.3b.

1.2 Software Architecture

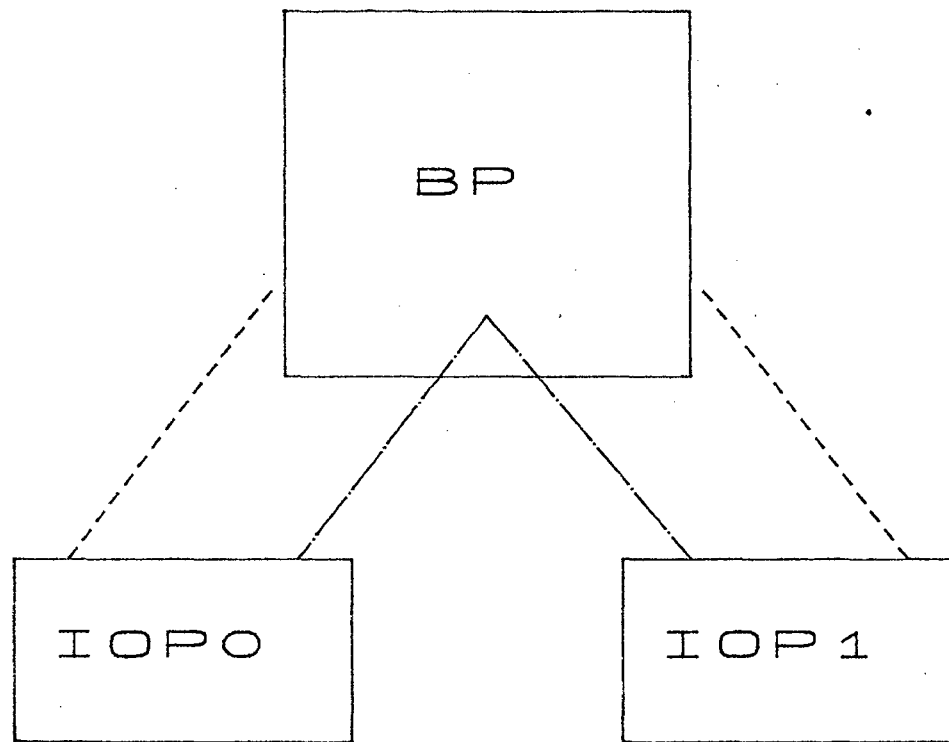
1.2.1 Overview

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 divided into a number of sub-systems. Fig 1.4 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

- C-DOT Real Time Operating System (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 different processors, through HDLC links. BM runs under CDOS.
- Call processing (CP) sub-system - is responsible for executing functions which actually process a call, e.g. call routing, call metering etc.
- Administrative (ADMN) sub-system - provides for management of exchange (billing, operator commands etc.) and other administrative support functions.
- Maintenance (MTCE) sub-system - provides functions for uninterrupted services to subscribers. This subsystem is responsible for detecting faults and recovering from them. It also provides for close monitoring of the systems sanity, comprehensive resource and data auditing facility which enables it to quickly detect software faults and prevent their propagation.
- Data base (DBMS) sub-system - manages the data bases globally. It uses a indexed sequential file management system (C-ISAM). It hides the physical organization of data from application programs.
- Peripheral processors (PP) sub-system - Actual telephony hardware (with 6502 CPU) is controlled by the sub-system.

1.2.2 Sub Systems Integration

Entire processing is distributed among these sub systems. They share and process global data and communicate with each other through different means of inter process (or



— . — . — . IOP-IOP LINK
— — — — IOP-BP LINK

FIG 1.5 HDLC LINKS FOR COMMUNICATION

inter processor) communication.

1.2.2.1 Inter Process Communication

Since different modules of different subsystems are functioning on different processors (in order to integrate them) inter-process (or inter-processor) communication becomes a vital aspect of software architecture. CDOS provides a uniform interface to all applications, so it becomes easy to communicate via CDOS. Fig. 1.5 shows how communication is performed between mate IOPs and between IOP and BP. (Here one would like to clarify that in SBM configuration IOP is connected to BM but in MBM it is connected to CM. Due to similarity in the architecture of CM and BM, for our purpose, we will not distinguish between the two). 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.

1.3 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 Uiscox, Bangalore telephone exchanges.

C programming language is used. C-DOT has its own data structures for various purposes, defined in various "C" header files [App. C]. Files and libraries are well organized and follow hierarchical 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 [10] depending upon various conditions such as which subsystem they are meant for, whether a file is process related or data related etc.

1.3.1 Why UNIX and 'C'

The UNIX operating system and "C" programming language are used for development as [7]

- UNIX is widely accepted standard operating system.
- Most of standard packages (as C-ISAM) are available on UNIX
- Highly portable system, runs on a variety of hardware.
- The structure of UNIX makes it adaptable to the applications needs.
- Inter-process communication is easily performed in UNIX
- and "C" provides a good interface to UNIX. It also on its own has powerful system programming facilities.

Chapter 2

Fault Tolerance

Fault tolerant computing is becoming more and more popular and indispensable as computer systems are being put to more and more complex applications. Literally fault tolerant system means, a system capable of recovering from a fault condition.

Providing uninterrupted power through battery backup to a computer system is a common example of a fault tolerant system. In order to achieve this two different sources of power are kept. In case main power supply fails, a switch over is performed from main supply to battery without any delay. The switch-over is performed by a system (electrical or electronic) which is capable of detecting a fault in main power supply. Similarly by introducing redundancy systems can be developed which are capable of detecting faults and recovering from them.

In certain systems, such as public switching system a condition of fault is not affordable. As B. Egert puts it, in connection with switching systems, in his paper titled Recovery Strategy for a Telephone Switching System [3]; "A very important factor in maintaining a telephone exchange is need to reach stringent reliability requirements by detecting and recovering from faults". Thus their design must incorporate features of fault tolerance. C-DOT DSS has been designed with this view in mind.

2.1 Fault Tolerance in C-DOT DSS

In all the basic modules, except IOP, duplication of hardware has been incorporated at card level. For example in BM, memory card and CPU card etc., are duplicated. At any instant one of them is active. In case of a fault in active card, a switch-over is performed from active to stand-by card without disrupting any of the ongoing processes. In the meanwhile factors responsible for the occurrence of fault can be cleared.

2.1.1 Duplex IOP Architecture

IOP, however, has been duplicated as a whole. Each IOP has its own set of I/O devices such as winchester disk, floppy disk, cartridge tape, printer, VDU etc. I/O devices of each IOP are not duplicated. Hence IOP with its own set of I/O devices forms a security block. IOPs are not configured as active/stand-by. Both IOPs function at any instant and are connected to each other via High Level Data Link Controller (HDLC). One of the IOP is designated as master IOP and other one as slave IOP. The master IOP is connected to BM (or CM) and is responsible for performing all the major functions. The slave IOP is tied to master IOP and all slave IOP - BM messages are routed through master IOP. Both the IOPs have their own set of duplicated data also. Backup devices like winchester are updated in both the IOPs; that is whenever an updation is performed on its winchester the master IOP requests the slave IOP to perform a similar updation on its own winchester. In case master IOP goes faulty, the slave IOP takes over the functions of master IOP. However a system is needed which is capable of detecting faults. Software faults are taken care of by maintenance software on IOP itself.

2.1.2 Maintenance Software On IOP

After the IOP's power is switched on, when UNIX is booted, maintenance sub-system's main process Input Output Configuration Manager (IOCM) takes the control. It schedules the other processes in the maintenance sub-system. It is responsible for resolving master/slave status of IOPs, resolving active/stand-by status of BPs etc. It stores

all information regarding statii etc. in IOP memory through semaphore operations (as provided by UNIX). It also monitors the overall sanity of IOP and if required changes the statii accordingly. It also ensures availability of up-to-date data to slave IOP by performing routine comparison (audit) of different sets of data present on IOPs. Audit Process is a part of maintenance software and is scheduled by IOCM.

2.2 Data Organization

C-DOT DSS supports distributed processing. BMs, running under CDOS perform actual switching. In their memory they have huge database. However for their mass storage requirements I/O devices connected to IOP are used. On IOP this data is stored in different files. As described earlier memory cards in BM are duplicated, the Base Processor (BP) writes on both the cards simultaneously, but reads from active card only. BP updates its database as and when required and updates the backup data residing in IOP files on an hourly basis.

Data on IOP is stored in different files. Files are further divided into different sets depending upon the kind of data they contain. Each IOP has same sets of files. Different file management systems are used to conveniently store different types of data.

2.2.1 File Management Systems

Although Unix offers only one kind of file, C-DOT DSS has two types of file systems.

2.2.1.1 UNIX files

As provided by UNIX. It has a distinct feature, a UNIX file is a consecutive stream of bytes 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. Truncation of files and selective modification is not permitted. 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. UNIX allows more than one users to access a file simultaneously; that is two or more process can simultaneously perform read/write operations on a file.

2.2.1.2 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. Appendix C 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.

2.3 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 process executes the required portion of program. As the 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))

Chapter 3

Audit Process

As discussed earlier, duplex IOP architecture requires routine auditing of duplicated sets of data present on IOPs. In addition to this the distributed processing architecture also requires some kind of mechanism to ensure consistency of different sets of global data available to different modules. This necessitates performing audits on sets of global data present on different basic hardware modules.

This chapter concentrates on different design requirements and considerations of audit process.

3.1 Audit Requirements

As has been discussed, two kinds of audits are required.

3.1.1 IOP - IOP (Disc to disc) audits

Essentially required by duplex architecture of IOP, these audits,, however also help in maintaining overall consistency of global data structures. These are performed by comparing corresponding files on IOPs. In case of inconsistency data is to be updated from master IOP.

3.1.2 IOP - BP (Disk to memory) audits

To ensure the availability of consistent data to different hardware modules such audits are performed. The data structures residing in BP memory are compared with corresponding files on IOP. In case of inconsistency data is to be updated from IOP file. IOP -BP audits are performed after IOP - IOP audits to ensure that data in IOP files is consistent.

3.2 Invocation

As discussed earlier audits should be performed periodically and on request. In C-DOT DSS audits can be invoked by different means.

3.2.1 System Initiated Auditing

Audits are invoked by system in case of any recovery. All unexpected errors encountered by processes lead to audits. Also whenever the system has reasons to suspect the data corruption or a data inconsistency, appropriate audits are to be performed. For example, in case of IOP switch over, there is a possibility of files generated or modified due to system initiated action to become inconsistent.

3.2.2 Calendar Based Routine Auditing

To run audits periodically, audit commands can be placed in calendar process. Calendar schedules these audits at specified frequency (say 10 Hr.).

3.2.3 Idle Time Audits

It is possible to schedule audits when the processor has no work at hand. All audits run during idle time depending upon frequency assigned to them.

3.2.4 Operator Initiated Audits

The operator can start audits by placing appropriate command. These are carried out

with highest priority, as it is assumed that operator will initiate audits when some kind of malfunctioning is noticed by him.

Keeping in mind the modular architecture of DSS software, it has been decided that all audit requests will be directed to Input Output Configuration Manager (IOCM), which after scheduling them will pass the requests to AUDIT PROCESS. Strictly speaking there is only one process, IOCM, which can initiate audits. However a report, containing the results of audits, number of discrepancies encountered etc. is sent to IOCM as well as to actual initiator of audits. Further, IOCM has the privilege of aborting the audits, at any stage. IOCM can also continue the audits from the stage it aborted.

3.3 Design Considerations

- Audits are prone to race conditions i.e.: by the time one gets the two sets of data, which need to be compared, one of them may get changed or it may be in a transient state. As a result audits may fail although the data may be consistent.
- Passing on the complete file or record, for comparison leads to heavy loading of communication links and thus should be avoided.
- Running of master functions of audits on master IOP will load it and may result in its performance degradation.
- Since there are some unused (dummy) bytes in data structures, checksum or even a byte by byte comparison may fail even though data structures may be consistent.
- Since IOCM schedules some other processes also, a provision should be there to abort the audits, whenever IOCM requires. Another provision for continuing the audits, from the point where audits were aborted, should

also be there.

- Files present on IOP are well arranged in different sub- directories. Nevertheless a provision should be there to change the path connecting these files without recompiling whole program. This will facilitate re-arrangement of files, if needed, at a later stage.
- Since there is no way to find the link status, in case of failure, audit may keep on waiting for a reply from its mate.
- The DSS configuration can have upto 32 BMs. Audit process should adjust itself according to actual number of BMs present in the configuration.
- Last but not the least, all messages should have a format as specified by C- DOT. Opcodes are defined in various header files and should be taken from there. However definitions which are exclusive to audits and are not required by any other process, can be defined locally.

3.4 Auditing Strategy

With above mentioned points following strategy was adopted for implementing audits.

3.4.1 IOP - IOP Audits

IOP - IOP audits can be performed at two levels. One which provides a gross check and other which provides an extensive check on data files. These are categorized as FILE LEVEL and RECORD LEVEL respectively.

3.4.1.1 File Level

It is possible to audit two files existing in IOPs. These audits will run periodically and also during idle time. However no corrective measure is taken based on this check. A failure

of this check leads to extensive Record Level check. If this check is satisfied then Record Level check is not done in case of routine and idle time audits.

The process of file level audits is as follows : The file on slave IOP is taken as slave file and file on master IOP is taken as master file. File size, in case of UNIX file and number of records in case of C-ISAM file, is computed by slave IOP and it directs the master IOP to perform the same function and reply back with results. Comparison is performed on slave IOP, after a reply is received from master IOP. If these checks pass, audits are said to be passed. The slave IOP will perform the task of report generation also.

3.4.1.2 Record Level

The process of record level auditing is as follows : The file on master IOP is taken as master file and file on slave IOP is taken as slave IOP as slave file. The slave file is scanned sequentially from the first record. Records are locked, a block of records is read and Cyclic Redundancy Code Checksum (CRC) of individual records are computed. To speed up the comparison process a CRC of all CRCs is also. computed. (size of block can be decided). Similar operation is carried out on master file, and the CRCs are passed on to slave IOP which carries out the task of comparing them. Individual CRCs are compared only when master CRC comparison fails. After detection of inconsistency, record is updated from master IOP. In case no discrepancy is found next block is read and process is repeated till end of file is reached. There is possibility of hitting following errors:

- Record is missing in slave IOP : Corresponding record from master IOP is ADDED in slave IOP file.
- CRC mismatch occurs : Corresponding record from master IOP file is COPIED on slave IOP file.
- Record is missing in master IOP : Extra record present in slave IOP file is DELETED.

- Audits fail in locking a record or file : This may be a temporary situation. Again n number of tries are given, if every try fails a report to this effect is sent to IOCM to enable it to take necessary action.

3.4.2 IOP - BP Audits

The data structures in BP memory are compared against the files existing on IOP. The IOP data is taken as master data, in case of any inconsistency. The record level locking concept holds good in this case also except that it is not possible to lock the data in BP memory. IOP - BP auditing is done after IOP - IOP auditing is over, to ensure that IOP has consistent data. Further IOP - BP audits have to take care of a number of BMs present in the configuration. The task of comparing CRCs etc. is done at BP end. On IOP end the process only retrieves the required data from files and pass the CRC/data to BP end. BP-end of IOP - BP audits has already been designed and implemented. To take care of variability in number of BMs present in the configuration following steps are taken:

3.4.2.1 Single Base Module

Since there is only one BM, one such process can take care of all such audit requests. In this case there is no need to provide BM number for communication, filename suffix etc.

3.4.2.2 Multi Base Module

In this case upto 32 BMs can exist and audit request can come from any of these BMs. The BM number is specified along with the request. A new copy of the process is executed for every request from a different BM. To achieve this a scheduler is written which schedules the audit requests and creates copies of the process for each BM.

Chapter 4

Implementation - I

In the previous chapter we discussed the strategy adopted for performing audits. This chapter concentrates on actual implementation of audits and attempts to describe its algorithm.

4.1 IOP - IOP Audits

The algorithm adopted in case of IOP - IOP audits is shown in fig 4.1. For convenience the algorithm has been divided in different portions. The reference to the *divided portion* appears in block letter with comments. Further, although a check is done for every kind of error, the steps are not shown in the algorithm. All kinds of errors, with a few exceptions, encountered lead to a function 'err_hand()' which handles these errors. This function is discussed in the next chapter. Errors which do not lead to this function are shown at the appropriate places.

```

program iopaudit
begin
  initialize the global variables;
  repeat
    receive a message;
    if IOP_status = master
    then
      begin
        FIGURE 4.1.1 (* perform slave functions *)
      end
    else
      begin
        FIGURE 4.1.2 (* perform master functions *)
      end
    end
  forever;
end (* of program *)

```

fig 4.1

```

(* perform slave functions *)

case ( opcode )
begin
  OPEN :
  begin
    open the requested file;
    reply back with the result; (* send message *)
  end
  CLOSE :
  begin
    close the requested file;
    reply back with the result; (* send message *)
  end
  READ :
  begin
    read the requested file; (* one record at a time *)
    reply back with the record; (* send message *)
  end
  CCRC :
  begin
    read the requested file; (* one block at a time *)
    compute CRC of all records;
    compute CRC of all CRCs; (* master CRC *)

```

```

        reply back with the result; (* send message *)
    end
    SIZE :
    begin
        if requested filetype = UNIX
        then
            begin
                find its size; (* number of bytes *)
            end
        else
            begin
                find number of records in it; (* C-ISAM files *)
            end (* fl *)
            reply back with the result; (* send message *)
        end
    end
    SEEK :
    begin
        move file pointer by requested amount; (* of the current
file *)

        reply back with the result; (* send message *)
    end
    others :
    begin
        do nothing; (* !!! *)
    end
end (* esac *)

```

fig. 4.1.1

(* perform master functions *)

```

case ( opcode )
begin
    START :
    begin
        extract set number;
        form stack of all the filenames; (* in the requested
set *)

    CONTINUE :
    begin
        do nothing;
    end
end

```

```

end
others :
begin
    set bad_opcode_flag;
end
end (* esac *)
while stack is not empty and no error flag is set
do
    read filename; (* from top of the stack *)
    read filetype; (* from top of the stack *)
    case ( subfield ) (* subfield of message received from IOCM
*)

```

```

begin
    FILE_LEVEL :
    begin
        FIGURE 4.1.2.1 (* file level audits *)
    end
    RECORD_LEVEL :
    begin
        FIGURE 4.1.2.2 (* record level audit algorithm *)
    end
end (* esac *)
if all flags are clear
then
begin
    pop (* bring stack pointer down by one *)
end (* fi *)
od
reply back with the result; (* to IOCM and to actual initiator
*)

```



TH-2879

fig 4.1.2

```

(* file level audits *)
if type = UNIX
then
begin
    find its size; (* number of bytes in the file *)

```

Dissertation .

681.3.06:621.3.06

Sa' 63

au

```

end
else
begin
    find number of records in it; (* C-ISAM file *)
end
set timer alarm; (* for time outs *)
send SIZE to mate; (* master IOP *)
receive a message;
if opcode = ABORT
then
begin
    set abort_flag;
end (* fl *)
if alarm
then
begin
    set time_out_flag;
end (* fl *)
if all flags are clear
then
begin
    compare the file sizes
    if file sizes are different
    then
    begin
        increment error_count;
    end (* fl *)
end (* fl *)

```

fig 4.1.2.1

```

(* record level audits *)
open the file;
set timer alarm; (* for time outs *)
send OPEN to mate; (* master IOP *)
receive a message;
if opcode = ABORT
then
begin
    set abort_flag;
end (* fl *)
if alarm
then
begin
    set time_out_flag;

```

```

end (* fl *)
if all flags are clear
then
begin
    FIGURE 4.1.2.2.1 (* compute CRC, compare CRC, update *)
end
close the file;
set timer alarm; (* for time outs *)
send CLOSE to mate; (* master IOP *)
receive a message;
if opcode = ABORT
then
begin
    set abort_flag;
end (* fl *)
if alarm
then
begin
    set time_out_flag;
end (* fl *)

```

fig 4.1.2.2

```

(* compute CRC, compare CRC, update *)
repeat
read a block;
compute CRC of all records;
compute CRC of all CRCs; (* master CRC *)
set timer alarm; (* for time outs *)
send CCRC to mate; (* master IOP *)
receive a message;
if opcode = ABORT
then
begin
    set abort_flag;
end (* fl *)
if alarm
then
begin
    set time_out_flag;
end (* fl *)
if all flags are clear
then
begin
    compare master CRC;

```



```

    if master CRCs are different
    then
    begin
        increment file_error_count; (* error count for this file
only *)

        if file_error_count > max_trials_to_be_given
        then
        begin
            update the file;
        else
        begin
            insert the file at bottom of the stack (* leave now, try
later *)

            end (* fi *)
        end (* fi *)
    end (* fi *)
until EOF or any flag is set;

```

fig 4.1.2.2.1

4.1.1 Explanation of algorithm

Audit process always keeps running (eternal process), it never exits on its own. After initializing the process it enters an infinite loop. It is now ready to receive a message from IOCM. After receiving the message it checks the master /slave status of the IOP. To avoid loading of master IOP, slave functions are executed on it while master functions are executed on slave IOP. [fig 4.1]

Slave functions, on master IOP, are tied to master functions. Only the requested function is executed and the result including errors encountered, are sent back to mate IOP. Opcodes are used to communicate. Subfield in the message is used to communicate the result of operation (FAILURE/SUCCESS). These opcodes are

- OPEN

- CLOSE
- READ
- CCRC
- SIZE
- and SEEK

Master functions are responsible for controlling slave functions in addition to performing their usual task of comparison etc. After receiving the message from IOCM, if IOP status is slave then these functions are executed. Two opcodes are expected from IOCM: START and CONTINUE.

In case of START a function is called to form a stack, implemented by a linked list, of all the filenames which belong to the requested set. In case of CONTINUE, however, such a stack already exists (remaining files of previous audits, which were aborted for some reasons) so only the `can_not_continue_flag` is checked to make sure that previous audits were aborted.

File attributes (name, type etc.) are read one by one, starting from the top of the stack and required audits (file level or record level) are performed. This process continues until either stack becomes empty (all files audited) or some error flag is set due to time-out or abort from IOCM etc. After this a report is sent to both IOCM and actual initiator of the audits.

File level audits are comparatively easier to implement as no corrective action is to be taken. For every file, the file size is computed, a request is sent to mate to perform a similar operation. Before going in wait state, expecting a message from mate, a timer alarm is set. This timer alarm will wake up the process in case mate is not able to reply. This is

done to avoid master functions waiting infinitely for a reply from mate.

At this point a check is also made for an ABORT message from IOCM. This provides IOCM, a mechanism by which it can stop the audits without killing the audit process. On reception of a 'SUCCESS' message from its mate, master function compares the file sizes. On discrepancy error count is incremented. In case 'FAILURE' is received file is inserted at the bottom of the stack. When rest of the files are finished this file will automatically come up for auditing.

Record level audits are slightly complicated. Unlike file level audits, many dialogues with mate IOP are required, in order to audit a file. First file is opened and after setting alarm, a message is sent to mate. It should be clarified here that whenever a dialogue with mate IOP is initiated, timer is set and after every dialogue message is checked for an ABORT. On receiving the reply from mate, a block is read and its CRCs calculated. The next message, to read a block and compute its CRC is sent. Received master CRC is compared against the one computed here. In case mismatch is found, no corrective is taken. Only a counter is incremented, which counts the number of times the discrepancy was found in the file. The process of reading a block, computing CRC etc. is repeated till end of file is reached. However, if this counter becomes greater than a pre-defined number, say 5, then a function is called to update the erroneous record from master file. This is done to ensure that discrepancies which were found in their transient states are not propagated. Only when on all occasions the comparison fails, the updation is performed.

4.2 IOP - BP Audits

The master functions of IOP - BP audits are residing on BP. On IOP only slave functions were developed. The algorithm is shown in fig 4.2. The opcodes have been defined globally, as a number of processes will be communicating with it.

```

program bpaudit
begin
  initialize;
  repeat
    receive a message;
    case ( opcode )
    begin
      MINTRO :
      begin
        extract the sender process' identification from the message;
        reply with own identification; (* introduction ! *)
      end
      MCRCRQ : (* request for computing CRC *)
      begin
        open the required file;
        read a block;
        compute CRCs of all the records;
        compute CRC of all the CRCs; (* master CRC *)
        reply back with CRCs;
      end
      MCRCFAIL : (* a mismatch has occurred *)
      begin
        read the required record;
        compute CRC of the record;
        reply back with record and its CRC;
      end
    end (* esac *)
  forever;
end (* of program *)

```

fig 4.2

4.2.1 Explanation

The IOP end of IOP - BP audits is tied to BP end. It performs only those functions which are requested by the BP end. Three different requests are expected. These are : MINTRO, MCRCRQ, MCRCFAIL.

MINTRO is received when some one initiates audits and BP-end starts talking with IOP-end. Process identifications are exchanged.

MCRCRQ is a request to read a particular block and compute its CRCs. These CRCs are then compared by BP-end. In case a discrepancy is conformed, BP-end sends a request (**MCRCFAIL**) to read the erroneous record and send it alongwith its CRC.

Validation

Initial tests are carried out on the machine used for developmental work (node IOPF in this case). All errors are removed. After succeeding in these tests the process is tried in lab, where an experimental setup is provided. After studying the behavior (such as CPU time consumed, amount of disk I/O done etc.) of the process for some time, the process is synchronized with other processes. When every thing goes fine it is tested on proper DSS configurations, either in Bangalore or in Delhi. The audits have been tried successfully on IOPF (one used for developmental work) and in the lab.

Chapter 5

Implementation - II

Simplicity, Upgradability and Maintainability are some of standard measures of software quality. While developing a software, programs should be modular so that they can be easily modified. Debugging modular software is a relatively easier task and is easier to understand by others. Modular design of software makes field debugging and upgrading, an easy task. Further, installation of patches becomes less tedious [5].

Keeping in mind these factors various functions were developed for implementation of AUDIT PROCESS. This chapter attempts to explain the different functions, algorithm adopted for functions is also discussed wherever necessary. For providing flexibility, all variables which are likely to change, have been defined as shell variables and then used inside the program. UNIX provides functions to manipulate the shell variables inside a 'C' program.

As audits are very sensitive in nature; i.e. any Error caused by audits may lead to fatal errors in the system. It is also important to handle error conditions in Audits very carefully. Any kind of modification in files should be done only when discrepancy is absolutely clear. Keeping this in mind, at every level extensive checks for various errors has been provided. All unexpected errors lead to abortion of audits and audit initialization.

5.1 IOP-IOP Audits

- `audit_init()`

Initialization is a very important task. Every flag is reset. Global variables like master/slave status of IOP are also initialized. This status is written in a semaphore by IOCM and is read from there. Message queue associated with audit is created, if required, and flushed. This is done to avoid reading spurious messages, if present in the queue. Pathname of files to be edited, is also read from shell variable "GLBDATAP". This step facilitates the re-arrangement of files at a later stage.

- `rcv_msg(buffer)`
- `send_msg(qid, buffer, nbytes)`
- `send(pid, buffer)`

For receiving and sending messages three functions have been written. All messages, including from BP and mate IOP are received on queue attached to audits. However messages across IOP are dispatched using `send` call of HDLC library and messages within IOP are dispatched using `msgsnd` call of UNIX.

- `extract(setno)`

For storing the attributes of all the files, on which audits are to be performed, a stack is implemented using linked list. A function has been written to extract all the required filenames (belonging to given set. set number 0 implies all files) and form a stack using linked list. Complete list of files present on IOP has been divided in different sets. A request for audits specifies whether all files are to be audited or a particular set (of files) is to be audited.

- push(stack number, item list)
- pop(stack number)
- insert(stack number, item list)

For manipulating the stack, three functions have been provided. These are :

- push: Which places an element on top of the stack.
- pop: which removes an element from top of the stack.
- insert: Which inserts an element at the bottom of the stack. Algorithm has

been derived from one described by D.E. Knuth [1].

- fil_open(name, type)
- fil_read(fd, type, buffer)
- fil_seek(fd, type, offset, whence)
- fil_crc(fd, type, nrecords, buffer)
- fil_close(fd, type)
- fil_size(fd, type, &recsize)

File operations require opening of files, with a proper check for locks, closing of file after releasing all locks, and reading the file record by record, after locking the records. For

these purposes different functions have been written.

fil_open: This function opens a file, by using ISAM function call isopen, in case of ISAM file. For UNIX file it executes system call open. Since UNIX does not provide for file locking, a kind of mutual exclusion technique is used. Existence of lock is indicated by the presence of a file by same name in directory specified by a shell variable. Before opening the required file, a check is made for existence of lock file. After successful opening of file, lock file is created in specified directory, to exclude others.

fil_close: It closes the required file, and releases all locks. UNIX file locks are released by deleting the file, created by fil_open. Releasing of locks is a very important operation. As locks exclude other processes from accessing the files, an unreleased lock can cause major damage to system sanity.

fil_read: This function reads a record of size recsize from a file identified by its file descriptor and type and fills the buffer with the data read.

fil_seek: Another very important file operation is to move around in a file. In case of UNIX lseek system call implements this, however, in case of C-ISAM files indirect approach had to be taken. In case of C-ISAM files the file pointer is moved according to the primary index only. Offset can be positive or negative. Pointer is moved in forward direction for positive offsets.

fil_size: Yet another file operation is to determine the number of records (bytes) in a C-ISAM file (UNIX). Function fil_size has been written for this purpose.

fil_crc: This function reads a block of data, calculates CRCs of all the records and CRC (master CRC) of all the CRCs. These CRCs are put in a buffer.

To calculate CRC of the data following function was developed.

- `crc(buffer, nbytes)`

```

program crc
begin
  initialize crc to 0;
  initialize byte pointer i=0;
  repeat
    set crc = exor two input bytes with crc;
    initialize bit counter;
    repeat
      shift the crc by 8 bits; (* shift *)
      set j = ( exor crc with FF Hex ) * ? (* mask and mul *)
      set k = jth element in look-up table;
      set crc = exor crc with k;
      increment bit counter;
    until bit counter > LIMIT (* limit hash defined *)
  until i > nbytes;
end (* of program *)

```

fig 5.1

The algorithm has been derived from the one (reduced table look up algorithm) explained in [2]. The values of shift, mul, mask and limit have been set after analyzing the performance of algorithm at different values.

- `update(mcrc_buff, srcr_buff, m_nrecords, s_nrecords)`

For comparing CRC and updating records whenever required, another procedure has been provided.

```

program update
begin
  nrecords = min( s_nrecords, m_nrecords )
  i = 0
  repeat
    if mcrc_buff[ i ] not = srcr_buff[ i ] (* mismatch pin pointed *)

```

```
then
begin
    move the pointer to ith record;
    set timer alarm;
    send SEEK to mate; (* bring mate's pointer to required place *)
    receive a message;
    if opcode = ABORT
    then
    begin
        set abort_flag;
    end (* fi *)
    if alarm
    then
    begin
        set time_out_flag;
    end
    if all flags clear
    then
    begin
        set timer alarm;
        send READ to mate; (* ask mate for the record *)
        receive a message;
        if opcode = ABORT
        then
        begin
            set abort_flag;
        end (* fi *)
        if alarm
        then
        begin
            set time_out_flag;
        end
        if all flags clear
        then
        begin
            if filetype = UNIX
            then
            begin
                write record; (* for UNIX file simply over-write the record *)
            end
            else
            begin
                rewrite the current record; (* try re-writing the record *)
                if rewrite fails; (* it fails if current record has different key value.. *)
            end
            end
        end
    end
end
```

```

then (* ..this means either a record is missing or is extra in master
begin
    write the record; (* try adding the record *)
    if write fails (* it may fail if record with this key value is present in sl.
    then
    begin
        delete current record; (* delete the extra record in slave file *)
    end (* fi *)
    end (* fi *)
end (* fi *)
end (* fi *)
end (* fi *)
end (* fi *)
increment i; (* look for discrepancy in the next record *)
until i > nrecords; (* until all records are checked *)
if s_nrecords < m_nrecords (* slave has some missing records *)
then
begin
    add all corresponding records from master file;
end (* fi *)
if s_nrecords > m_nrecords (* slave has extra records *)
then
begin
    delete all such records;
end (* fi *)
end (* of program *)

```

fig 5.2

It performs the task of updation. The strategy is as follows. In case of UNIX file the record from master file is copied or added as such for deletions, since UNIX does not allow for removing of bytes from a file, in indirect approach is used. A new temporary file is created with only required bytes. Then temporary file is moved in actual file name using 'C' function system(). In case of C-ISAM files first the record from master file is rewritten on erroneous record in slave file. If operation fails due to index duplication, which means a record is either extra or missing on slave file. Then record is written (added) on slave file, failure of this operation will indicate presence of an extra record on slave file which is then deleted. This process is repeated for every erroneous record.

err_hand(function, errorno)

Finally for handling different kinds of errors a procedure has been written. This function provides an extensive error checking mechanism. On every encounter of error, this function is called. This function analysis the error, in case of unexpected errors a check is made on master/slave status of IOP, on confirmation of a change, audits are aborted without sending a reply, otherwise abortion is accompanied by a report as usual. It also maintains a list and count of all errors encountered by audits.

5.2 IOP-BP Audits

Most of the functions required are common to both IOP - IOP and IOP - BP audits. Only those functions, which are exclusive to IOP - BP audits are discussed here.

- `extract_filename(index)`

A function is needed which can provide name of file, if index of that file is given.

- `block_seek(nblocks)`

Since in case of BP - IOP audits, CRC are to be computed block by block, a function for moving around in a file block by block has been developed over the function `fil_lseek()`.

Chapter 6

Conclusion

Audits were implemented and tested successfully. Audits involved lot of complexities and conceptual bugs besides programming bugs. In this chapter a critical review of AUDIT PROCESS is presented highlighting shortcomings and its solutions. Major factors determining quality of program are simplicity, sufficiency i.e. speed, and conceptual and implementation correctness. All of these are considered while making critical comments on a program.

6.1 Correctness of master file :

Here we have assumed that master file contains correct data. This assumption is not correct. As discussed earlier IOPs are not working in ACTIVE/STAND-BY configuration, Both IOPs are always active. One of them performing master functions and other one performing slave functions. In a situation when both IOPs are working a process may write on slave IOP first and then on master IOP. Some process may follow reversed sequence. In latter case master file will have reliable data, but in first case slave file will have reliable data.

A possible solution to this can be worked out if it can be ensured that all process will follow same sequence for, writing onto a file. Another approach could be to have triplicated data and thumb rule of, majority is authority is adopted.

However, both approaches involve a substantial increase in system overheads.

6.2 Deletion of record :

As far as question of deletion of record is concerned, It is always better to have a record, may be incorrect, than nothing at all !! Deletion of record may lead to complete loss of information. In case extra records are found in slave file these should be added in master file.

6.3 Locking :

To avoid synchronization problems locking is performed, this excludes other processes from accessing the file. Consider a process which locks a file for access and after locking is over, the operating system pre-empts the process. Now since file is locked, no other process can access the file and the process, which created the lock is waiting to be scheduled, by the operating system. In a situation, where locking is extensively used, the response time of all applications is bound to increase substantially. However a more alarming situation is, when after creating a lock, the process is killed due to, say, a bug in the software. In this case, the lock will never be released, which may lead to fatal blows to the system.

A solution can be found if concept of MONITORS [ref. 3] is introduced; that is there should be a process for performing read/write operations on a file. All other processes willing to perform read/write operation on a file should direct there request to this process, which in turn, will schedule them. The question of locking discussed in previous section can be taken care of, by this process.

However, all processes in the software system will need a change, this will involve several man-hours of programming.

6.4 Memory locks :

On BP side, the data resides in memory and there is no provision of locking the data in memory. Every kind of synchronization problem can be encountered in such an environment. The audit counterpart on BP side must take extra precautions, while updating a record. Introduction of memory locking will make BP processes slow and hence should be discouraged.

6.5 C-ISAM

The package used for handling C-ISAM files, is very slow. For manipulation of records it consumes a lot of time. Since C-ISAM calls are extensively used by audits, and other processes as well, these calls should be optimized and moved closer to the operating system kernel. This will result in an increase in overall response of the system. Further there is no equivalent of lseek system call of UNIX, which makes moving around in a C-ISAM file, without reading a record, highly inefficient. Providing such a function will make C-ISAM more powerful and side by side improve efficiency of application programs.

6.6 CRC

In audit process records are not compared byte by byte. The comparison is done by computing a Cyclic Redundancy Code for the records and comparing them. An inconsistency is said to have been encountered if CRCs are different, for the corresponding records. As has been observed, there are redundancies in CRCs; that is two altogether different records may yield same CRC. This property, may hide some discrepancies in the records, from the audits. If a 128 byte record is taken and a 2-byte CRC is computed, then on an average 64 different records may land up on same CRC. (we will refer to it as share ratio). Presently no solution exists for such a problem. However improved CRC algorithms can be adopted. For example, an algorithm which maps all records containing non ASCII data, onto a single CRC, will improve the share ratio for ASCII data, at the cost of non ASCII data. Such an algorithm is known as biased algorithm.

Appendix A

Abbreviations And Glossary

- 6502** — A micro processor with 8 bit data bus and 16 bit address bus.
- 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.
- AM** — Administrative Module: A basic module of C-DOT digital switching system.
- ASCII** — American Standard Code for Information Interchange.
- BIT** — Binary Digit.
- BM** — Base Module: Primary growth unit of C-DOT Digital Switching System and one of its four basic modules.
- BYTE** — A group of BITs (usually eight).
- C** — A programming language having powerful capabilities for system programming.
- CDOS** — C-DOT real time operating system: Base Processor with 68010 CPU

runs under CDOS and provides uniform interface to rest of the modules.

- C-DOT** — Centre for development of Telematics: India's Telecom technology Centre.
- C-ISAM** — A professional package for indexed sequential access file management system.
- 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.
- CPU** — Central Processing Unit
- CRC** — Cyclic Redundancy Code: A code used for transmission of data with a scope for error detection/correction.
- 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.
- 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.
- microVAX** — Micro VAX: a computer system developed by digital equipment corporation U.S.A.
- 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.
- OOS** — Out Of Service: Condition indicating that a unit is not working properly and has been removed from service.
- PP** — Peripheral Processing: Actual switching ie. making the physical connection between caller and called party, and other related functions fall under this category.
- 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.

- 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

System And Function Calls

B.1 UNIX System Calls

- `int creat (*filename, mode)`

Creates 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 decribed 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 positin 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.

- int msgget (key, flag)

Opens a queue tagged with a key value 'key'. if a queue with same key value does not exist and (flag & IPC_CREAT) is true then a new queue is created. Returns queue identifier of the queue or -1.

- int msgrcv (qid, *buffer, nbytes, mtype, msgflag)

Recieves first nbytes number of bytes of a message of type 'mtype' in buffer pointed by *buffer from a queue described by 'qid'. msgflag specifies whether to wait for a message or return immidiatly. Returns number of bytes recieved or -1.

- int msgsnd (qid, *buffer, nbytes, msgflag)

Sends nbytes number of bytes from a buffer pointed by *buffer with type of message

as 'mtype' in a queue described by qid. msgflag specifies whether to return immediately or keep on trying till message is actually sent. Returns 0 or -1.

- `int semget (key, number_sema, sem_flag)`

Opens or creates number_sems number of semaphores tagged with a key value 'key'. if a semaphore with same key value is not open and (sem_flag & IPC_CREAT) is true then semaphores are created. Returns semaphore identifier of the queue or -1.

- `int semctl (sid, snum, command, arg)`

Executes a variety of semaphore commands, specified by command and argument on semaphore described by semaphore identifier sid and semaphore number snum. Returns value as specified by command or -1.

- `int (*signal (sig, function)) ()`

Catches a signal specified by sig and performs function 'function'. Returns previous value of function or -1.

B.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 standered output file (stdout).

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

Scans the standerd input file (stdin) and accepts the values for arguments according to format as specified by control. Returns number of arguments succesfully 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)

Concatinates 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.

B.3 HDLC Library Function Calls

- `int addqid (qid, pid)`

Maps the given qid into CDOS type pid for communication through HDLC. Returns 0 or -1.

- `int send (pid, *buffer)`

Sends a message via HDLC to a process defined by pid from a buffer pointed by 'buffer'. Returns 0 or -1.

B.4 C-ISAM Library 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 whether next or previous or first or last or current record is to be read or a record 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 C

Source Code

C.1 The Organization

The complete source code has been organized in twelve different files. Two files 'mauditsys.c' and 'mauditfun.c' contain all the lower level functions. Main routine for IOP-IOP audits is in 'maudit.c', whereas main routine for BP-IOP audits is in 'bpiopaudit.c'. 'maudit.h' and 'bpiopaud.h' contain all the relevant hash and type defines. 'bpiopfun.c' contains functions which are exclusive to BP-IOP audits. A function has been developed for storing and retrieving the names of files present on IOP (ie. files to be audited) along with their attributes. This function is in file 'mauditdir.c' with 'mauditdir.h' containing all the relevant hash defines. A few macros developed as debugging tools are present in 'dbgmacro.h' with some global variables defined in 'dbgmacro.c'. Finally 'maudmacro.h' contains a few macros for IOP-IOP audits.

C.2 The Code

dbgmacro.h

```

/*
  Debugger Macros
*/
extern char *zerr[];
extern char *cont;
extern int  auderrno;

#define ADD_IN_FOR_MASTER_FAIL 200
#define AD_MASTER      ADD_IN_FOR_MASTER_FAIL

#define MIN_AUD_ERR  ( sys_nerr + 1 )
#define MAX_AUD_ERR  100

#define MIN_ERRNO  ( - HMAXERR )
#define MAX_ERRNO  is_nerr

#ifdef debug

#define Prnts(s)      fprintf(stderr, s)
#define Prntd(s,v)   fprintf(stderr,s,v)
#define Prntd1(s,v1,v2)  fprintf(stderr,s, v1, v2)
#define Pmterr(s)    perror(s)
#define CR()         fprintf(stderr,"n")

#define Prntmsg(no)  {\
  if ( (no) > MAX_ERRNO )\
  {\
    auderrno = (no)-MAXERRNO;\
    cont = " **On Master - %d";\
  }\
  else\
  {\
    auderrno = (no);\
    cont = " ** On Slave - %d";\
  }\
  fprintf(stderr, cont, auderrno);\
  if ( ( auderrno < 0 ) )\
  {\
    fprintf(stderr, " :-%s", hermmsg[ -(auderrno) ] );\
  }\
  else if ( auderrno < sys_nerr )\
  {\
    fprintf(stderr, " :-%s", sys_errlist[ auderrno ] );\
  }\
  else if ( auderrno < MAX_AUD_ERR )\
  {\
    fprintf(stderr, " :-%s", zerr[ ( auderrno)-MIN_AUD_ERR ] );\
  }\
  else if ( auderrno < is_nerr )\
  {\
    fprintf ( stderr, is_errlist[auderrno - 100] );\
  }\
}

#define Paray(start,len,fmt)  {\
  Int ii;\
  for ( ii=0; ii < (len); ii++ )\
    fprintf(stderr, fmt,*( (start)+ii));\
}

#else

#define Prnts(s)
#define Prntd(s,v)
#define Prntd1(s,v1,v2)
#define Pmterr(no)
#define CR()
#define Paray(start,len,fmt)
#define Prntmsg(n)

#endif

```

dbgmacro.c

```

int  auderrno;
char *cont;

#ifdef debug
char *zerr[20]= {
  "Job Over",
  "Aborted",
  "Bad Command",
  "Unkown Error",
  "No Such Set",
  "Time Out",
  "No Error",
  ".",
  ".",
  ".",
  ".",
  ".",
  ".",
  "Size Mismatch",
  "Bad Opcode",
  "Record Level",
  "CRC Mismatch",
  "File Level",
  "No Memory",
  "Master Fail"
};
#endif

```

mauditdir.h

```

#define MAX_FILES_IN_DIRECTORY (sizeof(dir) / sizeof(struct
directory))
#define MAX_FIL      MAX_FILES_IN_DIRECTORY

#define MAX_SET_NUMBER 3
#define MAX_SET      MAX_SET_NUMBER

#define MAX_TYPES_OF_FILES 2
#define MAX_TYP      MAX_TYPES_OF_FILES

/* File types */
#define UNIX 0
#define ISAM 1

extern struct directory  dir[];

```

mauditdir.c

```

/*
  Structure which contains information about directory
*/

struct directory  {
  Char *fname;
  Uint setno;
  Uchar type;
}  dir[] = {

  "dat"      , 1, UNIX,
  "dat1"     , 1, UNIX,
  "dat2"     , 1, UNIX,
  "idat"     , 2, ISAM,
  "idat1"    , 2, ISAM,
  "idat2"    , 2, ISAM,
  "DBID_LINE", 3, ISAM,
  "DBID_TRNK", 3, ISAM,

```

```

"DBID_CCKT", 3, ISAM,
"DBID_SREQ", 3, ISAM,
"DBID_TGOD", 3, ISAM,
"DBID_SGOD", 3, ISAM,
"DBID_CHRG", 3, ISAM,
"DBID_LNPF", 3, ISAM,
"DUID_SPRM", 3, ISAM,
"DUID_EXCD", 3, ISAM,
"DAID_DTOE", 3, ISAM,
"DAID_GPBK", 3, ISAM,
"DAID_HMGP", 3, ISAM,
"DAID_ETOS", 3, ISAM,
"DAID_DATB", 3, ISAM,
"DAID_LVL1", 3, ISAM,
"DAID_RTDS", 3, ISAM,
"DAID_OFTG", 3, ISAM,
"DAID_RTGP", 3, ISAM,
"DAID_PCTR", 3, ISAM,
"DAID_SCTR", 3, ISAM,
"DAID_TCTR", 3, ISAM,
"DAID_EQPD", 3, ISAM,
"DAID_RESF", 3, ISAM,
"DAID_SYDR", 3, ISAM,
"DBID_HDWR", 3, ISAM,
"DAID_PABX", 3, ISAM,
"DAID_TKGP", 3, ISAM,
"DAID_BMSS", 3, ISAM,
"DBID_HOUT", 3, ISAM,
"DBID_NOUT", 3, ISAM,
"DBID_HPVT", 3, ISAM,
"DBID_DOAM", 3, ISAM,
"DBID_DOA2", 3, ISAM,
"DBID_FOLO", 3, ISAM,
"DBID_DOFB", 3, ISAM,
"DBID_DOBS", 3, ISAM,
"DBID_ALARM", 3, ISAM,
"DBID_QDAT", 3, ISAM,
"DBID_HNGP", 3, ISAM,
"DBID_ABRD", 3, ISAM,
"DBID_SBRD", 3, ISAM,
"DBID_DNA2", 3, ISAM,
"DBID_DNAS", 3, ISAM,
"DBID_OPER", 3, ISAM };

```

maudmacro.h

```

/* status manipulating macros */

#define Opcode status_g.opcode
#define Ecount status_g.err_count
#define Gcount status_g.grand_e_count
#define State status_g.status_flag
#define Error status_g.error
#define Nfile status_g.files_left
#define Level status_g.level
#define Cset status_g.curr_setno

/* list manipulation macros */

#define Name top_g[0]->name
#define Type top_g[0]->type
#define Set top_g[0]->setno
#define Index top_g[0]->index
#define Tries top_g[0]->try
#define Recno top_g[0]->record_no

```

maudit.h

```

/* op-codes used internally, these are converted into op-codes
which are actually used to communicate with IOCM
*/

```

```

#define MSUCC 0
#define MFAIL -1

#define START_RL 1
#define START_FL 2
#define ABORT 3
#define CONTINUE_FL 4
#define CONTINUE_RL 5

#define OPEN 11
#define CLOSE 12
#define SIZE 13
#define READ 14
#define M_CRCC 15
#define LOC_RECORD 16
#define DELETE 17

```

```
/* Structures for communication with mate IOP */
```

```
/* Command to slave process on master IOP */
```

```
typedef struct {
    Hdr msg;
    Ulong offset;
    Ulong whence;
} CCommand;

```

```
/* Reply for OPEN */
```

```
typedef struct {
    Hdr msg;
    Uchar result;
    Ulong error;
    Uint rec_size;
    Uint id;
} OPen;

```

```
/* Reply for CLOSE */
```

```
typedef struct {
    Hdr msg;
    Uchar result;
    Ulong error;
    Uint id;
} CClose;

```

```
/* Reply for SIZE */
```

```
typedef struct {
    Hdr msg;
    Uchar result;
    Ulong error;
    Ulong size;
    Ulong no_of_rec;
} SSize;

```

```
/* Reply for READ */
```

```
typedef struct {
    Hdr msg;
    Uchar result;
    Ulong error;
    Uint rec_len;
    Uchar crc;
    Char record[1];
} REad;

```

```
/* Reply for CCRC */
```

```
typedef struct {
    Hdr msg;
    Uchar result;
    Ulong error;
    Uint no_of_records;
    Uint rec_len;
    Uint last_rec_len;
    Ushort master_crc[1];
} CCrc;

```

```
/* Reply for SEEK */
```

```

typedef struct {
    Hdr msg;
    Uchar result;
    Ulong error;
    Ulong seeked;
    Ulong current;
} SEek;

/* Audit result */
typedef struct {
    Uchar opcode;
    Uint err_count;
    Uint grand_e_count;
    Uchar status_flag;
    Ulong error;
    Uint files_left;
    Uchar level;
    Uint curr_setno;
} SStatus;

/* error codes in addition to UNIX, HDLC and C-ISAM errors */

#define SIZE_FAIL (MIN_AUD_ERR + 11)
#define BAD_OPCODE (MIN_AUD_ERR + 12)
#define RL (MIN_AUD_ERR + 13)
#define CRC_FAIL (MIN_AUD_ERR + 14)
#define FL (MIN_AUD_ERR + 15)
#define NO_MEMORY (MIN_AUD_ERR + 16)
#define MASTER_FAIL (MIN_AUD_ERR + 17)

/* op-code definitions for actual communication with IOCM */

#define JB_OVR (MIN_AUD_ERR + 1)
#define ABRTD (MIN_AUD_ERR + 2)
#define BAD_COMMAND (MIN_AUD_ERR + 3)
#define UNKNOWN (MIN_AUD_ERR + 4)
#define NOSUCH_SET (MIN_AUD_ERR + 5)
#define TIME_OUT (MIN_AUD_ERR + 6)
#define NO_ERROR (MIN_AUD_ERR + 7)
#define BUSY (MIN_AUD_ERR + 8)

/* opcodes used by mmu, pop, push, insert */
#define INIT 01
#define ALLOC 02
#define FREE 03

/* defines to be used for CRCC computations */

#define SHIFT 4
#define MUL 1
#define LIMIT 2
#define MASK 0xf

/* max defines */

#define MAX_RECORDS 128

#define MAX_UNIX_RECORD_SIZE 128
#define M_RSIZE MAX_UNIX_RECORD_SIZE

#define MAX_UNIX_FILENAME 14
#define MAX_NAME MAX_UNIX_FILENAME

#define MAX_PATH_STRING_LENGTH 256
#define MAX_PATH MAX_PATH_STRING_LENGTH

#define MAX_FILE_NAME_STRING (MAX_PATH + MAX_NAME)
#define MAX_FNAME MAX_FILE_NAME_STRING

#define MAX_TRY 3

/* Buffer size definitions */

#define COMMUNICATION_BUFFER_SIZE 512
#define S_CBUFF COMMUNICATION_BUFFER_SIZE

```

```

#define CRC_BUFFER_SIZE ((MAX_RECORDS - 1) <
1)
#define S_CRBUFF CRC_BUFFER_SIZE

#define READ_BUFFER_SIZE 256
#define S_RBUFF READ_BUFFER_SIZE

#define STACK_BUFFER_SIZE 2048
#define S_SBU#F STACK_BUFFER_SIZE

/* Structures for storing various variables */
typedef Char FName[ MAX_FNAME ];

typedef struct
{
    FName name;
    Int type;
    Int setno;
    Int index;
    Int error;
    Int try;
    Ulong record_no;
    struct List *next;
} Llist;

```

mauditsys.c

```

#ifdef INCLUDE
#include <sys/types.h>
#include <stdio.h>
#include <errno.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "u/tools/rel1_13/systyp.h"
#include "audit.h"
#include "auditext.c"
#include <isam.h>
#include "dbgmacro.h"
#endif

/* Routine which initializes the audit process
* Initialization includes
* 1. Flushing of message receive queues.
* 2. Finding pid of mate, for hdic send by addqid().
* 3. Finding master/slave status by maslave().
* 4. Resetting of all flags.
* 5. Path string by getenv().
* 6. Own Pid by gethostid().
*/

Int audit_init()
{
    Int qid;

    Prmts("[audit_init]: ");

    /* Find mate and own Pid */
    addqid(selfid_g,AUD_ID);
    own_pid_g = gethostid();
    mate_pid_g = own_pid_g ^ 0x01;
    Prmt1("/* gethostid returns ownid = %d, mateid = %d */",
    own_pid_g, mate_pid_g);

    /* Reset all flags */
    master_fail_flag_g = 0;
    filelevel_g = 0;
    abort_flag_g = 0;
    inconsis_count_g = 0;

```

```

#ifdef IOPF

printf("Enter mas, rkey, skey : ");
scanf("%d %d %d", &master_g, &self_key_g, &skey_g);

#else

/* Determine master/slave status of IOP */
if ( ( master_g = maslave () ) == MFAIL )
    return ( MFAIL );
Prntd("{ master_g = %d } ", master_g);

#endif

/* Flush message queues */
while (iop_receive(self_key_g, c_buff_g, S_CBUFF,
IPC_NOWAIT) != MFAIL);

/* Path string */
if ( ( pathname_g = getenv( "GLBDATAP" ) ) == MFAIL )
    return ( MFAIL );
Prntd(" getenv returns path = %s */ ", pathname_g );

/* initialize linked list */
mmu ( stack_g, sizeof ( stack_g ), INIT );

Prnts("-initialised");
return ( MSUCC );
}

/* Routine which finds master/slave status of IOP */

Int maslave()

{

Int val;
Int semid;

Prnts("[ maslave ]: ");

/* Find semaphore id */
if ( ( semid =
semget(IOP_STATUS_SEM_KEY,NO_SEMS,IPC_CREAT|
0666) ) == MFAIL )
    return ( MFAIL );
Prntd(" semget returns semid_g = %d */ ", semid_g );

if ( ( val = semctl ( semid_g,MAST_SLV_SEM_NUM,GETVAL,0) )
== MFAIL )
    return ( MFAIL );
Prntd(" semctl returns val = %d */ ", val );

if ( val == SEM_VAL_MASTER )
    master_g = 1;
else
    master_g = 0;

return ( MSUCC );
}

/*
* Routine which copies the given number of bytes from
* source to destination and terminates it with null byte.
*/

Void stcpy (dest,src,num)

Char *dest;
Char *src;
Int num;

{
while ( num !=0)
{
*dest++ = *src++;
--num;
}
}

```

```

push(no,el,err,tr)

Int no;
Int el;
Int err;
Int tr;

{

Prnts("[push]: ");
if( (ptr_mai=mmu ( NULL, S_SBUFF, ALLOC)) != NULL )
{
ptr_mai->element = el;
ptr_mai->next=root_mai[no];
ptr_mai->error=err;
ptr_mai->try=tr;
root_mai[no]=ptr_mai;
}
else
Prnts("-FATAL no memory abandoning this file ");
}

insert(no,element,error,try)

Int no;
Int element;
Int error;
Int try;

{

Prnts("[insert]: ");
if( (ptr_mai=mmu ( NULL, S_SBUFF, ALLOC)) != NULL )
{
ptr_mai->element=element;
ptr_mai->next=base_mai->next;
ptr_mai->error=error;
ptr_mai->try=try;
base_mai->next=ptr_mai;
base_mai=ptr_mai;
}
else
Prnts("-FATAL : no memory abandoning this file ");
}

pop(no)

Int no;

{

Prnts("[pop]: ");
if (root_mai[no] != NULL )
{
ptr_mai=root_mai[no]->next;
mmu (root_mai[no], 0, FREE);
root_mai[no]=ptr_mai;
}
}

/* Routine which does the memory management for stack
implementation */

struct Llist *mmu ( buffer, bufsize, func )
struct Llist *buffer;
Int bufsize;
Uchar func;

{
static struct Llist *top;
static struct Llist *top_sys;
static struct Llist *sys_pointer;

struct Llist *tmp;

Prntd("[ mmu ]: ");
switch ( func )
{
case INIT :

```

```

if ( buffer == NULL )
    return ( NULL );
top_sys = NULL;
top = buffer;
tmp = top+sizeof ( struct Llist );
for(; tmp<=buffer+buffsize; tmp+=sizeof( struct Llist ))
    {
        top->next = tmp;
    }
top->next = NULL;
top = buffer;
return ( top );

case ALLOC :
if ( top != NULL )
    {
        if ( top_sys == sys_pointer )
            {
                sys_pointer = top_sys = NULL;
                free ( sys_pointer );
            }
        tmp = top;
        top = top->next;
        return ( tmp );
    }
else
    {
        if ( sys_pointer == NULL )
            {
                if((top_sys=mmu(malloc(buffsize),buffsize,INIT))==
NULL )
                    return ( NULL );
                sys_pointer = top_sys;
            }
        tmp = top_sys;
        top_sys = top_sys->next;
        return ( tmp );
    }

case FREE :
if ( sys_pointer == top_sys )
    {
        buffer->next = top;
        top = buffer;
        return ( buffer );
    }
else
    {
        buffer->next = top_sys;
        top_sys = buffer;
        return ( buffer );
    }
}

/* This routine returns the qid, if required it creates a
 * queue.
 * arg : key
 * returns : qid or -1
 */

#define MAXOPEN 10

Int    openqueue(key)

Long    key;
{
static    struct {
Long key;
Int qid;
} queues[MAXOPEN];
Int    i,avail,qid;

Prntd("[openqueue]: -key %d ",key);
avail = -1;
for ( i = 0; i< MAXOPEN; i++) {
    if (queues[i].key == key)
        return (queues[i].qid);
    if (queues[i].key == 0 && avail == -1)
        {
            avail = i;
            break;
        }
}
if (avail == -1)
    {
        Prnts("-too many queues open ");
        return ( MFAIL );
    }
if ((qid = msgget (key,0666|IPC_CREAT)) == -1)
    {
        Prnterr("-msgget fails -");
        return ( MFAIL );
    }
Prntd("{ msgget returned qid=%d } ",qid);

queues[avail].key = key;
queues[avail].qid = qid;
return (qid);
}

/*
 * Routine which performs the send from the IOP
 */
Int    iop_send(dstkey,buf,nbyte) /* send message */
Long    dstkey;
MSgbuf *buf;
Int    nbyte;
{
Int    qid;
Int    ret;

Prntd("[iop_send]: -nbytes %d ",nbyte);
if ((qid = openqueue(dstkey)) == MFAIL )
    {
        return( MFAIL );
    }
Prntd("/ qid=%d */",qid);
buf->mtype = 1;
if ( (ret = msgsnd(qid,buf,nbyte,-IPC_NOWAIT)) == MFAIL )
    {
        Prnterr("-msgsnd fails-");
        return (MFAIL);
    }
return(ret);
}

/*
 * Routine which performs the receive message function.
 * arg : key, buffer, number of bytes, wait/nowait
 * returns : number of bytes actually read or -1.
 */
Int    iop_receive(srckey,buf,nbyte,nowait)

Long    srckey;
Char    *buf;
Int    nbyte,nowait;
{
Int    qid;
Int    ret;

Prntd("[iop_rcv]: -nbytes %d ",nbyte);
if ((qid = openqueue(srckey)) == MFAIL )
    {
        return ( MFAIL );
    }
Prntd("{ qid=%d } ",qid);

if ( (ret = msgrcv(qid,buf,(nbyte-4),0,nowait) ) == -1 )
    {
        Prnterr("-msgrcv fails -");
    }
}

```



```

        return( MFAIL );
    }
}
Prntd("/* msgrcv rcvd %d bytes */", ret);
return(ret);
}

/* This routine determines the size (size of record) of a
UNIX ( C-ISAM ) file. It also fills no. of complete
records in 'second'.
arg : filename, type of file.
returns : size ( no of records ) of the file or -1
*/

Int size( fname, type, second)

Char    *fname;
Int     type;
Int     *second;

{
    struct stat    buf;
    struct dictinfo dic;
    Int           i;

    Prnts("[size]: ");
    if ( type )
    {
        if ( (i=stat(fname,&buf)) == MFAIL )
        {
            Prnterr("-stat fails -");
            return ( MFAIL );
        }
        Prntd("/* stat shows size=%ld */", buf.st_size);
        *second = buf.st_size / M_RSIZ;
        return( buf.st_size );
    }
    else
    {
        if ( (i=isopen(fname,ISMANULOCK+ISINPUT) == -1 )
        {
            Prnterr(iserrno);
            return ( MFAIL );
        }
        if ( isindexinfo(i, &dic, 0) == MFAIL )
        {
            Prntmsg(iserrno);
            return ( MFAIL );
        }
        isclose ( i );
        Prntd("/* indexinfo shows nrecords=%d */",dic.di_nrecords);
        *second = dic.di_nrecords ;
        return ( dic.di_recsize );
    }
}

/* This routine opens a file in read-only mode
arg : file-name , type of file
returns : file identifier or MFAIL in case of error
*/

Int fil_open(fname,type)

Char    *fname;
Int     type;

{

    Int     f_id;
    struct keydesc key;
    struct dictinfo dic;

    Prnts("[fil_open]: ");
    if ( type )
    {
        if ( (f_id=open(fname,O_RDONLY)) <= 0 )
        {
            Prnterr("-open fails -");
            return ( MFAIL );
        }
    }
}

```

```

        rec_size_mai = 128;
        Prntd("/* fd = %d */",f_id);
        return( f_id);
    }
    else
    {
        if ( (f_id=isopen(fname, ISMANULOCK + ISINOUT) ) ==
MFAIL)
        {
            Prntmsg ( iserrno );
            return ( MFAIL );
        }
        Prntd("/* isfd = %d */",f_id);
        if ( isindexinfo( f_id, &dic, 0) == MFAIL )
        {
            Prntmsg ( iserrno );
            return ( MFAIL );
        }
        Prntd("/* rec-size = %d */",dic.di_nrecords);
        rec_size_mai = dic.di_recsize;
        if ( isindexinfo( f_id, &key, 1) == MFAIL )
        {
            Prntmsg ( iserrno );
            return ( MFAIL );
        }
        if ( isstart( f_id, &key, 0, 0, ISFIRST ) < 0 )
        {
            Prntmsg(iserrno);
            return ( MFAIL );
        }
        return ( f_id );
    }
}

/* This routine closes an already open file
arg : file identifier , type of file
returns : MFAIL in case of failure or id of closed file.
*/

Int fil_close(f_id,type)

Int     f_id;
Int     type;

{
    Prnts("[fil_close]: ");
    if ( type )
    {
        return( (close(f_id)== -1) ? MFAIL : f_id);
    }
    else
    {
        return ( (isclose ( f_id)== -1 ) ? MFAIL : f_id );
    }
}

/* This routine reads in specified number of bytes from a file
arg : file identifier , type of file , pointer to buffer
pointer to buffer for crc values
returns : number of records read - 0 in case of EOF
MFAIL in caes of failure
*/

Int fil_read(f_id,type,buff,crcbuff,rec_size,fun)

Int     f_id;
Int     type;
Char    *buff;
Ushort *crcbuff;
Uint    rec_size;
FLg     fun;

{

    Int     er, j;
    Int     count;
    Uint    no;

    Prnts("[fil_read]: ");
}

```

```

if ( type)
{
    if ( fun )
        no=rec_size ;
    else
        no=buff_size ;
    if ( (er = read(f_id,buff,no) < 0 )
        {
            Prnterr("-read fails -");
            return ( MFAIL );
        }
    Prntd("/ read %d bytes */",er);
    if ( fun )
        {
            if ( er < no )
                buff[er]=0;
            return ( er );
        }
    bytes_read_mai=er;
    if ( er == 0 )
        {
            crcbuff[0]=0;
            return ( 0 );
        }
    j=( (er - 1)/rec_size) + 1;
    for (count=1; count<=j; count++)
        {
            if ( er >= rec_size )
                tmp_mai=rec_size;
            else
                tmp_mai=er;
            er -= tmp_mai;
            crcbuff[count] = cre(buff,tmp_mai);
            buff += tmp_mai;
        }
    crcbuff[0]=cre(&crcbuff[1],2*(count-1));
    Prntd("-master crc=%u ",crcbuff[0]);
    if ( count < 63 )
        crcbuff[count]=0;
    return( (count-1) );
} /* if type = UNIX */
else
{
    for ( j=1; j<64; j++)
        {
            if ( isread ( f_id, buff, ISNEXT ) != SUCC )
                {
                    switch ( iserrno )
                    {
                        default :
                            {
                                Prntmsg(iserrno);
                                return ( MFAIL );
                            }
                        case EENDFILE :
                            {
                                if ( fun )
                                    {
                                        buff[er]=0;
                                        return ( 0 );
                                    }
                                break; /* from for loop */
                            }
                    }
            Prntd(" %d",j);
            if ( fun )
                {
                    if ( rec_size < max_path_len )
                        buff[rec_size]=0;
                    return ( rec_size );
                }
            crcbuff[j]=cre ( buff, rec_size );
        }
    bytes_read_mai=j*rec_size;
    tmp_mai=rec_size;
    if ( j < 63 )
        crcbuff[j]=0;
    crcbuff[0]=cre ( &crcbuff[1], 2*(j-1) );
    Prntd("-master crc=%ud ", crcbuff[0]);
    return ( j-1 );
} /* else */
}

```

```

/*
routine to move around in a c-isam file specified by fid.
buffer is filled with the record.
arg      : fid, offset, buffer
returns  : no of records actually moved, -1 if fail
*/
Int islseek( offset, fid, buffer)

Int      offset;
Int      fid;
Char     *buffer;

{
    Int      i;

    Prntd("[islseek]: -offset %d ", offset);
    if ( isread ( fid, buffer, ISCURR ) == MFAIL )
        if ( iserrno != EENDFILE )
            {
                Prntmsg(iserrno);
                return ( MFAIL );
            }
    if ( offset == 0 )
        {
            return ( 0 );
        }
    if ( offset < 0 )
        {
            for ( i=0; i<= -(offset) ; i++)
                if ( isread ( fid, buffer, ISPREV ) == MFAIL )
                    break;
            return ( i-1 );
        }
    if ( offset > 0 )
        {
            for ( i=0; i< (offset) ; i++)
                if ( isread ( fid, buffer, ISNEXT ) == MFAIL )
                    break;
            return ( i-1 );
        }
}

```

mauditfun.c

```

#ifdef INCLUDE
#include <sys/types.h>
#include <stdio.h>
#include <errno.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include "/u/tools/rel1_13/systyp.h"
#include "audit.h"
#include "auditext.h"
#include "dbgmacro.h"
#include "dbgmacro.c"
#endif

/* This routine computes the crc of a given record
*/
Ushort cre(input,nlength)

Uchar *input;
Int nlength;

{
    static Ulong crc_table[] = { 0x0,0x1081,0x2102,0x3183,
                                0x4204,0x5285,0x6303,0x7387,0x8408,0x9489,0xa50a,
                                0xb58b,0xc60c,0xd68d,0xe70e,0xf78f };
}

```

```

Ushort pinp;
Ushort inicrc, mask;
Uchar i, mul, limit, shift;

shift = SHIFT;
limit = LIMIT;
mask = MASK;
mul = MUL;
inicrc = 0;
for ( ; nlength > 0 ; nlength-- )
    {
    ((Uchar*)&pinp)[0] = *input++;
    ((Uchar*)&pinp)[1] = *input++;
    inicrc ^= pinp;
    for ( i=0; i<limit; i++ )
        inicrc = ( inicrc > shift ) ^
            crc_table[ mul * (mask & inicrc) ];
    --nlength;
    }
return(inicrc);
}

/* This is error routine */

Int err_hand(i,j)

Uint i;
Uint j;
{

Prmts("[err_hand: ]");
switch(i)
    {

    case -1 :

        while (root_mai[1]!=NULL)
            {
            Prmtd(" %s ",directory[root_mai[1]->element].fname);
            Prmtmsg(root_mai[1]->error);
            pop(1);
            }
        break;

    case 2 :
    case 1 :

        if ( root_mai[0]!= NULL )
            {
            if ( j==SIZE_FAIL )
                {
                root_mai[0]->try=MAX_TRY;
                }
            if ( master_fail_flag_mai ) j=msg_mai.extra+150;
            if ( root_mai[0]->try++ >= MAX_TRY )
                {
                Prmts("-too many tries, abandoning it.");
                push(1,root_mai[0]->element,j,root_mai[0]->try);
                }
            else
                {
                Prmts("-to be tried later. leaving it.");
                insert(0,root_mai[0]->element,j,root_mai[0]->try);
                }
            if ( i != 2 )
                {
                pop(0);
                CR();
                }
            }
        break;

    default :

        Prmtmsg(i);
        Prmtmsg(j);
    } /* case */

```

```

}

/* Routine which reads a semaphore and returns the present
 * master slave status
 */

Int maslave( semid )

Int semid;

{
Int val;

if ( (val = semctl( semid_g,MAST_SLV_SEM_NUM,GETVAL,0))
== MFAIL )
    return ( MFAIL );
return ( val );
}

/* routine to initialize the global variables */

/* This routine sends the message to the mate iop
arg: destination pid, buffer address, # of
bytes.
returns : 0 or -1
*/

Int send_msg( pid, buffer, nbytes )

Ulong pid;
Char *buffer;
Long nbytes;

{
Int j;
Long dest;

Prmts("[send_msg: ]");
#ifdef IOPF
( ( Hdr * )buffer )->pid = own_pid_g ;
dest = pid;
#else
( ( Hdr * )buffer )->pid = ( Long )own_key_g ;
dest = pid;
#endif
buffer = (MSgbuf *)&msg_mai;
msg_mai.mtype=0;
msg_mai.opcode=code;
msg_mai.flag = typ;
j = sizeof(i->opcode) + sizeof(i->flag) + sizeof(i->extra);
strcpy(msg_mai.name, filename,nbytes);
j += nbytes + sizeof(i->nbytes);
msg_mai.nbytes=j;
if ( (j=iop_send(skey,buffer,j)) == MFAIL )
    {
    return ( MFAIL );
    }
Prmt1("-leng %d type %d ",msg_mai.nbytes,msg_mai.flag);
return(MSUCC);
}

/* This routine recieves the message.
arg : pointer to the message buffer
returns : opcode in the message or -1.
*/

Int rcv_msg ( buffer )

Char *buffer;

{

```

```

Int      opcode, j, type, actual;

Prmts("rcv_msg:");
if ( ( j = iop_receive ( own_key_g, buffer, S_CBUFF, 0 ) ) ==
MFAIL )
{
    return ( MFAIL );
}
Prmtd("{ bytes read = %d ", j);

actual = ( ( Hdr * )buffer )->dummy;
if ( j != actual )
{
    Prmtd("-msg bytes lost, actual = %d ", actual );
    return ( MFAIL );
}

opcode = ( ( Hdr * )buffer )->opcode;
type = ( ( Hdr * )buffer )->subfield;

Prmtd1("-opcode %d type %d -", opcode, type );
Paray( buffer + sizeof ( struct Hdr ), 5, "%x ");
return( opcode );
}

/* This routine extracts index to be audited from data
structure containing details of files by forming a linked
list ( root ).
arg : setno.
returns : no of files extracted, In case no file is extracted
MFAIL is returned
*/

Int extract(set)

Int      set;

{
    Int      all;
    Int      index=0;
    Int      count=0;

    Prmtd("[extract ]: -setno %d ", set);
    all=set;
    if ( !set )
        set++;
    while ( set <= max_setno )
    {
        for (index=0; index<=max_index; index++)
            if ( directory[index].setno==set )
            {
                Prmtd("-file= %s ",directory[index].fname);
                count++;
                push( 0, index, 0, 0);
            }
        if ( all )
            break;
        set++;
    }
    if (root_mai[0]==NULL)
    {
        Prmts("-no file in this set");
        return( MFAIL );
    }
    base_mai=root_mai[0];
    while (base_mai->next != NULL)
        base_mai=base_mai->next;
    return( count );
}

/* routine which sends the report to IOCM and to initiator
process whose pid it recieved from IOCM.
arg      : pid, address of message buffer.
returns  : MSUCC or MFAIL

Int report ( pid, buffer )

??? pid;

```

```

Char *buffer;
{
    * Routine which sends the message to the master IOP
    * giving the result of the audit function.

Void s_mate(result,rec)
Int result;
Int rec;
{
    m_ptr_g = iomap_malloc(sizeof(struct lnkmsg));
    l_ptr_g = (struct lnkmsg*)m_ptr_g;
    l_ptr_g->mten = (sizeof(Mprfmd));
    ((Mprfmd*)(l_ptr_g->info))->msg.opcode = MPRFMD;
    ((Mprfmd*)(l_ptr_g->info))->msg.subfield = 01;

    ((Mprfmd*)(l_ptr_g->info))->msg.sndr.pr_id.mtce_byte=OPPHY;
    ((Mprfmd*)(l_ptr_g->info))->msg.sndr.pr_id.module_num=mate_id_g

    ((Mprfmd*)(l_ptr_g->info))->msg.sndr.pr_id.process_num=IOMAP_I
    dest_id_g = ((Mprfmd*)(l_ptr_g->info))->msg.sndr.id;
    ((Mprfmd*)(l_ptr_g->info))->msg.sndr.pr_id.module_num=self_id_g;

    ((Mprfmd*)(l_ptr_g->info))->result = result;
    ((Mprfmd*)(l_ptr_g->info))->record_num = rec;
    send (dest_id_g,m_ptr_g);
    free (m_ptr_g);
}
*/

```

maudit.c

```

#ifdef INCLUDE
#include <stdio.h>
#include "/u/tools/rel1_13/systyp.h"
#include <isam.h>
#include <errno.h>
#include "audit.h"
#include "dbgmacro.h"
#include "mauditdir.h"
#endif

/* storage space allocation */

/* Data structure for maintaining the linked list which holds
information about files to be audited
*/

List      *ptr_g, *top_g[2]=(NULL,NULL), *base_g;

/* Different global variables, buffers and flags
*/

Uchar      master_fail_flag_g;
Uchar      abrt_flag_g;
Uchar      filelevel_g;
Uchar      master_g;

Char      combuff_g[ S_CBUFF ];
Char      *cbuff_g = (char *) ( ((lnkmsg *)combuff)->info );
Ushort    crcbuff_g[ S_CRBUFF ];
Char      rbuff_g[ S_RBUFF ];

Int      setno_g;
Char      *fname_g;
Char      *pathname_g;
FName     filename_g;
Int      f_id_g;

```

```

Int     type_g;
Uint    index_g;
Int     rec_size_g;
Int     last_rec_size_g;
Int     cur_ptr_g;

#ifdef IOPF
Int     sndr_qid_g;
Int     mate_qid_g;
#else
Pid     sndr_pid_g;
Pid     mate_pid_g
#endif
Pid     own_pid_g;

Int     self_key_g = AUD_ID;
Int     iocm_key_g = IOCM_ID;

STatus  status_g;

main()
{
    Int     opcode;
    Char    *tmpbuf;
    Int     tmpsize;
    Uint    level;
    Uint    setno;

    Int     i,j,temp,k;
    FLg     file_flag;
    Int     result0,result1;
    Long    rec_red;
    Int     read_flag;

    Prnts("[main]: -initialising ");

    status_g.grand_e_count = 0;
    while ( audit_init() != MFAIL );
    CR();

    while ( 1 )
    {
        /* For testing on iops compile with -DIOPF option */
#ifdef IOPF
        if ( master_g )
        {
            if ( (opcode = rcv_msg( cbuff_g )) == MFAIL )
            {
                /* it should report to its own IOCM */
                audit_init();
                continue;
            }
        }
        else
        {
            printf("Enter code, level, setno : ");
            scanf("%d %d %d", &opcode, &level, &setno );
        }
#else
        if ( (opcode = rcv_msg( cbuff_g )) == MFAIL )
        {
            /* it should report to its own IOCM */
            audit_init();
            continue;
        }
#endif

        #endif

        if ( master_g )
        {
            Int test;

            Prnts("Master: ");

```

```

/* What else can it do ??? */
if ( ( ( COmmand * )cbuff )->msg.pid != mate_pid_g )
    continue;*/

switch ( opcode )
{
    Long tmp;
    Long second;

    case SIZE :

        type_g = ( ( COmmand * )cbuff )->msg.subfield;
        index_g = ( ( COmmand * )cbuff )->msg.dummy;
        Prntd1("-SIZE type= %d, index= %d, name= ", type_g,
index_g);
        fname_g = directory_g[ index_g ].name;
        strcpy ( filename_g, pathname_g );
        strcat ( filename_g, fname_g );
        Prnts(filename_g);

        if ( (temp=(Long)size( filename_g, type_g, &second) )
== MFAIL )
        {
            ( ( Size * )cbuff )->result = MFAIL;
            ( ( Size * )cbuff )->error = ( type_g ? errno : iserrno );
        }
        else
        {
            ( ( Size * )cbuff )->result = MSUCC;
            ( ( Size * )cbuff )->size = temp;
            ( ( Size * )cbuff )->no_records = second;
        }
#ifdef IOPF
        test=send_msg(mate_qid_g, cbuff_g, sizeof(struct
Size));
#else
        ( (Inkmsg *)combuff_g )->mten = sizeof ( struct Size );
        test = send( mate_pid_g, combuff_g );
#endif
        break;

    case OPEN :

        type_g = ( ( COmmand * )cbuff )->msg.subfield;
        index_g = ( ( COmmand * )cbuff )->msg.dummy;
        Prntd1("-SIZE type= %d, index= %d, name= ", type_g,
index_g);
        fname_g = directory_g[ index_g ].name;
        strcpy ( filename_g, pathname_g );
        strcat ( filename_g, fname_g );
        Prnts(filename_g);

        if ( (temp=fil_open( filename_g, type_g )) == MFAIL )
        {
            ( ( OPen * )cbuff )->result = MFAIL;
            ( ( OPen * )cbuff )->error = ( type_g ? errno :
iserrno );
        }
        else
        {
            ( ( OPen * )cbuff )->result = MSUCC;
            ( ( OPen * )cbuff )->id = temp;
            ( ( OPen * )cbuff )->rec_size = rec_size_g;
        }
#ifdef IOPF
        test=send_msg(mate_qid_g, cbuff_g, sizeof(struct
OPen));
#else
        ( (Inkmsg *)combuff_g )->mten = sizeof ( struct OPen );
        test = send( mate_pid_g, combuff_g );
#endif
        break;

    case CLOSE :

        type_g = ( ( COmmand * )cbuff )->msg.subfield;
        fid_g = ( ( COmmand * )cbuff )->msg.dummy;
        Prntd1("-SIZE type= %d, fid= %d ", type_g, fid_g);

        if ( (temp=fil_close( fid_g, type_g )) == MFAIL )
        {
            ( ( CLose * )cbuff )->result = MFAIL;

```

```

iserrno);    (( CClose *)cbuff)->error = ( type_g ? errno :
              }
              else
              {
                (( CClose *)cbuff)->result = MSUCC;
                (( CClose *)cbuff)->id = temp;
              }
#ifdef IOPF
CClose));    test=send_msg(mate_qid_g, cbuff_g, sizeof(struct
#else
              (( lnkmsg *)combuff_g)->mflen = sizeof ( struct CClose );
              test = send( mate_pid_g, combuff_g );
#endif
              break;

              case READ :

              type_g = (( CCommand *)cbuff)->msg.subfield;
              fid_g = (( CCommand *)cbuff)->msg.dummy;
              Prntd1("-SIZE type= %d, fid= %d ", type_g, fid_g);

              tmpbuf = ((REAd *)cbuff)->record;
              if((temp=fil_read(fid_g,type_g,tmpbuf))==MFAIL)
              {
                (( REAd *)cbuff)->result = MFAIL;
                (( REAd *)cbuff)->error = ( type_g ? errno :
iserrno );
              }
              else
              {
                (( REAd *)cbuff)->result = MSUCC;
                (( REAd *)cbuff)->rec_len = temp;
                (( REAd *)cbuff)->crc = cre(tmpbuf, temp);
              }
              tmpsize = sizeof( struct REAd ) + temp - 1;
#ifdef IOPF
              test=send_msg(mate_qid_g, cbuff_g, tmpsize);
#else
              (( lnkmsg *)combuff_g)->mflen = tmpsize;
              test = send( mate_pid_g, combuff_g );
#endif
              break;

              case CCRC :

              type_g = (( CCommand *)cbuff)->msg.subfield;
              fid_g = (( CCommand *)cbuff)->msg.dummy;
              Prntd1("-SIZE type= %d, fid= %d ", type_g, fid_g);

              tmpbuf = ((CCrc *)cbuff)->master_crc;
              if((temp=fil_crc(filename_g,type_g,tmpbuf))==MFAIL)
              {
                (( CCrc *)cbuff)->result = MFAIL;
                (( CCrc *)cbuff)->error = ( type_g ? errno : iserrno
);
              }
              else
              {
                (( CCrc *)cbuff)->result = MSUCC;
                (( CCrc *)cbuff)->rec_len = rec_size_g;
                (( CCrc *)cbuff)->last_rec_len = last_rec_size_g;
                (( CCrc *)cbuff)->no_of_records = temp;
              }
              tmpsize = sizeof( struct CCrc ) + ((temp - 1) < 1);
#ifdef IOPF
              test=send_msg(mate_qid_g, cbuff_g, tmpsize);
#else
              (( lnkmsg *)combuff_g)->mflen = tmpsize;
              test = send( mate_pid_g, combuff_g );
#endif
              break;

              case SEEK :

              type_g = (( CCommand *)cbuff)->msg.subfield;
              fid_g = (( CCommand *)cbuff)->msg.dummy;
              off = ((CCommand *)cbuff)->offset;
              from = ((CCommand *)cbuff)->whence;
              Prntd1("-SIZE type= %d, indefid= %d ", type_g, fid_g);
              if( (temp=fil_seek(fid_g, type_g, off, from)) == MFAIL)
              {
                (( SEek *)cbuff)->result = MFAIL;
                (( SEek *)cbuff)->error = ( type_g ? errno : iserrno
);
              }
              else
              {
                (( SEek *)cbuff)->result = MSUCC;
                (( SEek *)cbuff)->seeked = temp;
                (( SEek *)cbuff)->current = cur_ptr_g;
              }
#ifdef IOPF
              Open));    test=send_msg(mate_qid_g, cbuff_g, sizeof(struct
#else
              (( lnkmsg *)combuff_g)->mflen = sizeof ( struct OPeN );
              test = send( mate_pid_g, combuff_g );
#endif
              break;

              } /* case struct */
              CR();
              Prntd("send result = %d ", test);
              CR();
              continue; /* while ( 1 ) */
            }
          else
          {
#ifdef IOPF
#else
            opcode = cbuff->msg.opcode;
            level = cbuff->msg.subfield;
            setno_g = cbuff->msg.dummy;
#endif
            status_g.opcode = opcode;
            status_g.err_count = 0;
            status_g.status_flag = BUSY;
            status_g.setno = setno;
            status_g.error = 0;
            status_g.level = level;

            audit_slave (opcode, level, setno_g, &status_g );
            err_hand(result0,result1);
            Prnts("report follows ");
            /* report(); */
            err_hand(-1,-master_mai);
            CR();
            CR();
          } /* while ( 1 ) */
        } /* main */

        /* slave procedure */
        void audit_slave ( opcode, level, setno, report )
        Int opcode;
        Int level;
        Int setno;
        SStatus *report;
        {
          Int ret = JB_OVR;
          Int no_of_file;
          Int i;
          Int j;
          Int result;

          Prnts("[audit_slave]: ");
          switch ( opcode )
          {
            case START :

```

```

top_g[0]=NULL;
top_g[1]=NULL;
if ( ( report->files_left = extract( setno ) )==MFAIL)
{
    report->status_flag = ABRTD;
    report->error = NOSUCH_SET;
    return;
}
Prntd("extracted %d files ",report->files_left);
CR();
break;

case CONTINUE :

if ( report->files_left )
{
    report->status_flag = ABRTD;
    report->error = NO_FILES_LEFT;

    return;
}
break;

default :

report->status_flag = ABRTD;
report->error = BAD_COMMAND;

return;
}
while ( top_g[0]!= NULL )
{
    index_g = top_g[ 0 ] -> element;
    type_g = dir[ index_g ].type;
    fname_g = dir[ index_g ].fname;
    report->curr_setno = dir[ index_g ].setno;
    Prntd1("-index= %d -type= %ld ",index_g, type_g);
    strcpy ( filename_g, pathname_g );
    strcat ( filename_g, fname_g );
    Prnts(filename_g);

    switch ( level )
    {

        case FL :

            if( (result = perform(SIZE)) == MFAIL)
            {
                if ( report->status_flag )
                {
                    report->error = ABORT;
                    return;
                }
                err_hand(1, ( type ? erno : iserno ));
                continue;
            }
            if ( result != cbuff->size )
            {
                err_hand(1,SIZE_FAIL);
                report->error = SIZE_FAIL;
                report->err_count++;
                report->grand_e_count++;
                continue;
            }
            break;

        case RL :

            if( ( f_id_g =perform(OPEN))==MFAIL)
            {
                if ( report->status_flag )
                {
                    report->error = ABORT;
                    return;
                }
                err_hand(1,(type? erno : iserno));
                continue;
            }
            if ( (result = comp_updt ( f_id_g )) == MFAIL )
                report->error = CRC_FAIL;;
            if ( ( result = perform(CLOSE))==MFAIL)

```

```

{
    err_hand(1, (type ? erno : iserno));
    continue;
}
if ( !report->status_flag )
{
    report->files_left--;
    pop(0);
}
else
{
    report->error = ABORT;
    return ;
}

break;

default :

report->error = BAD_COMMAND;
report->status_flag = ABRTD;
return;
}

CR();
CR();
} /* while */
return ;
}

/* This routine compares and if required, updates the file
described by id */

Int comp_updt ( id )
Int id;

{
    while ( ((result=perform(READ))!=MFAIL) && !abrt_flag_mai )
    {
        if ( msg_mai.flag == 0 )
            break;
        if ( crcbuff[0] != *((Ushort *)&msg_mai.name[0]))
        {
            Prnts("-mismatch found ");
            if(root_mai[0]->try < MAX_TRY)
            {
                err_hand(2,CRC_FAIL);
                break;
            }
        }
        /*update(type,&rcbuff[1],&msg_mai.name[2],sizeof(msg_mai.flag,msg
        _mai.extra, f_id, fname),
        if((j==MFAIL)&&(root_mai[0]->try<MAX_TRY))
        {
            err_hand(2,CRC_FAIL);
            break;
        }
        ret = CRC_FAIL;
    }
}

/* This routine performs the task of a given opcode including
send & rcv message ,check for failure at both ends
arg : opcode
returns : result of the fun. if every thing is fine else MFAIL
*/

Int perform(opcode)

Int opcode;

{
    Long temp,temp1;
    Int master_fail_flag =0;

    Prntd("[perform]: -opcode %d ",opcode);
    switch(opcode)
    {
        case OPEN :

```

```

temp1=fil_open(isname,type);
break;

case READ :

temp1=fil_read(f_id,type,buff,crcbuff,rec_size_mai,0);
msg_mai.extra= rec_size_mai ;
break;

case CLOSE :

temp1=fil_close(f_id,type);
break;

case DELETE :

case LOC_RECORD :

msg_mai.extra=tmp_mai+1;
temp1=MSUCC;
break;

case SIZE :

if ( ! type )
{
len_isname = strlen (isname);
}
temp1 = size ( isname, type );
break;

}
if (temp1 == MFAIL && opcode != CLOSE )
{
Prnts("-operation fails ");
return ( MFAIL );
}
else
{
if ( send_msg(opcode,isname,type,len_isname) == MFAIL )
{
Prnts("-cannot send message ");
return( MFAIL );
}
/* time outs */
alarm ( 30 );
signal( SIGALRM, timeout );
if ( (temp=rcv_msg(&msg_mai)) == MFAIL )
{
Prnts("-cannot receive message ");
return( MFAIL );
}
alarm ( 0 );
if ( msg_mai.opcode != opcode )
{
/* really no sol. */
abrt_flag_mai=1;
Prnts("-opcode mismatch, aborting audit ");
return ( MFAIL );
}
switch(msg_mai.opcode)
{
case ABORT : /* any need? */

Prnts("-abort recieved, aborting audit ");
abrt_flag_mai=1;
return ( MFAIL );

default :

if (msg_mai.flag == MFAIL)
{
master_fail_flag_mai=1;
Prnts("-operation failed on master ");
return ( MFAIL );
}
}
return ( temp1 );
} /* else */
}

```

```

Int update(type,slave,mastr, rec_s, rec_m, last_rec, fid, filename)

Int      type;
Ushort  *slave, *mastr;
Long     rec_s , rec_m ;
Int      last_rec;
Int      fid;
Char     *filename;

{
Ushort  crcbuff_s[max_path_len], crcbuff_m[max_path_len];
Int      max, min;
Int      i, last_s;
FLg      flag;
Char     bu;
Int      fd;
Int      pos;
Char     command[max_path_len+18];
FLg      write_flag;
Char     tbuff[max_path_len<];
Int      index_s =0 ;
Long     num;

Prnts("[update]: ");
strcpy(crcbuff_s, slave, sizeof(crcbuff_s));
strcpy(crcbuff_m, mastr, sizeof(crcbuff_m));
if ( type )
{
if( (pos=lseek(fid, 0, 1))== -1)
{
Prnterr("-lseek fails -");
return( MFAIL );
}
strcpy(command, "mv why_no_del.unx ");
flag = ( rec_m < rec_s );
if ( ( pos=lseek(fid,-bytes_read_mai, 1) ) != bytes_read_mai )
{
Prnterr("-lseek not proper -");
return (MFAIL );
}

for ( tmp_mai=0; tmp_mai< (rec_m-1) ; tmp_mai++ )
{
if ( crcbuff_s[ tmp_mai ]!=crcbuff_m[ tmp_mai ] )
{
if ( (i=perform(LOC_RECORD))==MFAIL )
{
Prnts("-cannot update ");
return(MFAIL);
}
}
if(write(fid, msg_mai.name, rec_size_mai) !=
rec_size_mai)
{
Prnterr("-write fails, cannot update -");
return( MFAIL );
}
}
else
if (lseek(fid, rec_size_mai, 1) == -1)
{
Prnterr("-Lseek fails -");
return( MFAIL );
}
}
if ( crcbuff_s[ tmp_mai ]!=crcbuff_m[ tmp_mai ] )
{
if ( (i=perform(LOC_RECORD))==MFAIL )
{
Prnts("-cannot update last record ");
return(MFAIL);
}
}
if(write(fid, msg_mai.name, msg_mai.flag) !=
msg_mai.flag)
{
Prnterr("-write fails on last record, cannot update
-");
return( MFAIL );
}
last_s=bytes_read_mai;
while ( last_s > rec_size_mai )

```



```

        last_s -= rec_size_mai;
        if ((rec_s == rec_m) && (last_rec < last_s))
            flag=1;
    }
    else
        if (lseek(fid, pos, 0) != pos)
            {
                Prnterr("-lseek fails -");
                return (MFAIL);
            }
        if (flag)
            {
                i=lseek(fid, 0, 1);
                if ((fd=creat("why_no_del.unx", 0644)) < 0)
                    {
                        Prnts("-cannot truncate file on slave");
                        return(MFAIL);
                    }
                lseek(fid, 0, 0);
                while (i-- > 0)
                    {
                        read(fid, &bu, 1);
                        max=write(fd, &bu, 1);
                    }
                close ( fd );
                close ( fid );
                strcat(command, filename);
                system(command);
            }
        return ( MSUCC );
    }
    else
        {
            flag=( rec_m != rec_s );
            pos = isrecnum;
            for ( i=0; i <= rec_s; i++)
                {
                    if ( isread ( fid, tbuff, ISPREV ) < 0 )
                        {
                            if ( iserrno != EENDFILE )
                                {
                                    Prntmsg(iserrno);
                                    return ( MFAIL );
                                }
                        }
                }
            tmp_mai =0;
            while ( (index_s < rec_s) && (tmp_mai < rec_m) )
                {
                    if (isread ( fid, tbuff, ISNEXT ) != MSUCC )
                        {
                            if ( iserrno != EENDFILE )
                                {
                                    Prntmsg(iserrno);
                                    return( MFAIL );
                                }
                        }
                    write_flag = 1;
                }
            num = isrecnum;
            if ( crcbuff_s[ index_s ] != crcbuff_m[ tmp_mai ] )
                {
                    while ( 1 )
                        {
                            if ( index_s >= rec_s || tmp_mai >= rec_m )
                                break;
                            if ( (i=perform(LOC_RECORD)) == MFAIL )
                                return ( MFAIL );
                            if ( iswrite ( fid, msg_mai.name ) == MFAIL )
                                {
                                    if ( iserrno != EDUPL )
                                        {
                                            Prntmsg(iserrno);
                                            return ( MFAIL );
                                        }
                                }
                            else
                                {
                                    tmp_mai++;
                                    continue;
                                }
                        }
                }
            while(isrewrec(fid,num,msg_mai.name)==MFAIL)

```

```

            {
                if ( iserrno != EDUPL )
                    {
                        Prntmsg(iserrno);
                        return ( MFAIL );
                    }
                if (isdelrec (fid,num) == MFAIL)
                    {
                        Prntmsg(iserrno);
                        return (MFAIL);
                    }
                index_s++;
                if ( isread( fid, tbuff, ISNEXT ) == MFAIL )
                    {
                        Prntmsg(iserrno);
                        return ( MFAIL );
                    }
                num = isrecnum ;
            }
            break;
        }
        index_s++;
        tmp_mai++;
    } /* while loop */
    if ( tmp_mai < rec_m )
        if ( perform(LOC_RECORD) == MFAIL )
            return ( MFAIL );
    if ( flag )
        {
            if ( index_s < rec_s )
                while ( isread( fid, tbuff, ISNEXT ) == MSUCC )
                    isdelrec ( fid, isrecnum );
            else
                while ( perform (LOC_RECORD) != MFAIL )
                    {
                        tmp_mai++;
                        if (iswrite (fid, msg_mai.name) == MFAIL)
                            {
                                Prntmsg(iserrno);
                                return (MFAIL);
                            }
                    }
        }
    return ( MSUCC );
}

void timeout();
{
status_g.status_flag = 1;
}

```

bpiopaud.h

```

/* Buffer sizes */
#define COMMUNICATION_BUFFER_SIZE 512
#define S_CBUFF COMMUNICATION_BUFFER_SIZE

#define READ_BUFFER_SIZE 256
#define S_RBUFF READ_BUFFER_SIZE

/* Data structures for communication are defined in ??
*/

#define MFAIL -1
#define MSUCC 0
#define ISAM 0

#define OWN_KEY 4
#define IOCM_KEY 5
#define BP_AUDIT_ID 10

/* block size */

```

```
#define BLOCK_SIZE 2

/* Max defines */
#define MAX_UNIX_FILE_NAME 14
#define M_UNAME MAX_UNIX_FILE_NAME

#define MAX_PATH_STRING 256
#define M_PSTRING MAX_PATH_STRING

#define MAX_FILE_NAME_STRING (M_UNAME + M_PSTRING)
#define M_FNAME MAX_FILE_NAME_STRING

#define MAX_RECORD_SIZE_UNIX 128
#define UR_SIZE MAX_RECORD_SIZE_UNIX
```

bpiopaud.c

```
Int  mintro_s = sizeof ( struct Mintro );
Int  mioprec_s = sizeof ( struct Mioprec );
Int  mcrcrep_s = sizeof ( struct Mcrcrep );
Int  lnkmsg_s = sizeof ( struct lnkmsg );
Int  mintro_s = sizeof ( struct Mintro );
Int

Char  kk_g[ S_RBUFF ];
Char  *rbuffer_g = kk_g;
struct {
    int  mlen;
    Char  ki_g[ S_CBUFF ];
}  msgbuff_g;
Char  *cbuffer_g = msgbuff_g.ki_g;

Int  own_key_g = OWN_KEY;
Int  iocm_key_g = IOCM_KEY;

Pid  own_pid_g;
Pid  sndr_pid_g;

Ulong  rec_size_g;

Char  kp_g[ M_PSTRING ];
Char  *path_g = kp_g

Char  ku_g[ M_UNAME ];
Char  *uname_g = ku_g;

Char  kf_g[ M_FNAME ];
Char  *fname_g = kf_g;

main()
{
    Int  opcode;
    Int  dsize;
    Ushort  crc;
    Ulong  rnum;
    Int  index;
    Char  *pntr;

    bpaud_init();
    while ( 1 )
    {
        if ( ( opcode = rcv_msg ( msgbuff_g ) ) == MFAIL )
        {
            Prnts("rcv_msg fails\n");
            exit(2);
        }
        switch ( opcode )
        {
            case MINTRO :

                sndr_pid_g = ( ( Mintro * ) cbuffer_g )->msg.sndr;
#ifdef IOPF
                send_msg ( sndr_pid_g, cbuffer_g, mintro_s );
#else
                ((Mintro *)cbuffer_g)->msg.sndr = own_pid_g;
#endif
            }
        }
    }
}
```

```
msgbuff_g.mlen = sizeof( struct lnkmsg );
send ( sndr_pid_g, msgbuff_g );
#endif

break;

case MCRCRQ :

    index = ((Mcrcrq *)cbuffer_g)->file;
    length = ( (Mcrcrq *) cbuffer_g )->length;
    bp_no = ( (Mcrcrq *) cbuffer_g )->msg.pr_id.module_num;
    blk_no = ((Mcrcrq *) cbuffer_g )->block_no;

    if ( ( uname_g = extract_fname(index) ) == NULL )
    {
        Prnts("NO such Index");
        exit(3);
    }

    strcpy ( fname_g, path_g );
    strcat ( fname_g, uname_g );
    /* BP number concatenation ?? */
    /* type of file ?? */
    if ( ( fid_g = fil_open ( fname_g, type_g ) ) == MFAIL )
    {
        Prntd(" can not open %s ", fname_g );
        exit ( 4 );
    }

    Prntd("{ fid = %d } ", fid_g );

    block_seek ( fid_g, blk_no );

    sndr_pid_g = ( ( Mcrcrq * ) cbuffer_g )->msg.sndr;
#ifdef IOPF
    send_msg ( );
#else
    ((Mcrcdata *)cbuffer_g)->msg.sndr = own_pid_g;
    msgbuff_g.mlen = sizeof( struct lnkmsg );
    send ( sndr_pid_g, msgbuff_g );
#endif

    break;

case MCRCFAIL :

    sndr_pid_g = ( ( Mcrcfail * ) cbuffer_g )->msg.sndr;

    index = ((Mcrcfail *)cbuffer_g)->file;
    bp_no = ( (Mcrcfail *) cbuffer_g )->msg.pr_id.module_num;
    blk_no = ((Mcrcfail *) cbuffer_g )->block_no;

    if ( ( uname_g = extract_fname(index) ) == NULL )
    {
        Prnts("NO such Index");
        exit(3);
    }

    strcpy ( fname_g, path_g );
    strcat ( fname_g, uname_g );
    /* BP number concatenation ?? */
    /* type of file ??? */
    if ( ( fid_g = fil_open ( fname_g, ISAM ) ) == MFAIL )
    {
        Prntd(" can not open %s ", fname_g );
        exit ( 4 );
    }

    block_seek( fid_g, blk_no, ISAM );
    pntr = &( ( (Mioprec *) cbuffer_g )->data );
    if ( ( rnum = fil_read ( fid_g, pntr, ISAM ) ) == MFAIL )
    {
        Prntd(" can not read %d ", fid_g );
        exit ( 5 );
    }

    if ( ( fid_g = fil_close ( fid_g, ISAM ) ) == MFAIL )
    {
        Prntd( "can not close %d ", fid_g );
    }

    crc = cre ( pntr, rsize_g );
    pntr = ( (Mioprec *) cbuffer_g )->index;
    fill_index ( fid_g, ISAM, pntr );
    /* index ????? */
    dsize = sizeof ( struct Mioprec ) - 1 + rsize_g;
    /* -1 for field 'data[1]' in Mioprec */
    ((Mioprec *)cbuffer_g)->dat_size = rsize_g;
    ((Mioprec *)cbuffer_g)->crc = crc;
```

```

#ifndef IOPF
    send_msg ();
#else
    msgbuff_g.mlen = dsize;
    ((Mioprec *)cbuffer_g)->msg.sndr = own_pid_g;
    send ( sndr_pid_g, msgbuff_g );
#endif
    break;

    } /* end case */
} /* end while */
} /* end of main */

```

bpiopfun.c

```

#ifndef INCLUDE
#include <stdio.h>
#endif

/* Routine which returns a pointer to name of the file with
given index.
Arg      : index
Returns  : pointer to filename.
*/

char *filename( index )

int index;

{
    static char *name[] = {
        "DBID_LINE",
        "DBID_TRNK",
        "DBID_CCKT",
        "DBID_SREQ",
        "DBID_TGOD",
        "DBID_SGOD",
        "DBID_CHRG",
        "DBID_LNPF",
        "DUID_SPRM",
        "DUID_EXCD",
        "DAID_DTOE",
        "DAID_GPBK",
        "DAID_HMGP",
        "DAID_ETOS",
        "DAID_DATB",
        "DAID_LVL1",
        "DAID_RTDS",
        "DAID_OFTG",
        "DAID_RTGP",
        "DAID_PCTR",
        "DAID_SCTR",
        "DAID_TCTR",
        "DAID_EQPD",
        "DAID_RESF",
        "DAID_SYDR",
        "DBID_HDWR",
        "DAID_PABX",
        "DAID_TKGP",
        "DAID_BMSS",
        "DBID_HOUT",
        "DBID_NOUT",
        "DBID_HPVT",
        "DBID_DOAM",
        "DBID_DOA2",
        "DBID_FOLO",
        "DBID_DOFB",
        "DBID_DOBS",
        "DBID_ALRM",
        "DBID_QDAT",
        "DBID_HNGP",
        "DBID_ABRD",
        "DBID_SBRD",
        "DBID_DNA2",
        "DBID_DNAS",
        "DBID_OPER" };
}

```

```

if ( index < 0 || index > 44 )
    return ( NULL );
return ( name[ index ] );
}

/* Routine which places the file pointer at the given block.
arg :   file descriptor, blockno, type of file.
return : no of records moved or -1.
*/

int block_seek ( fid, blk, type )

int fid;
Ushort blk;
Ushort type;

{
    int ret;

    if ( type )
    {
        if ( ( bytes = lseek ( fid, bytes, 0 ) ) == MFAIL )
        {
            Prnterr(" can not lseek ");
            return ( MFAIL );
        }
        return ( ret );
    }
    else
    {
        if ( ( rec = islseek ( fid, rec ) ) == MFAIL )
        {
            Prnterr("can not is-lseek ");
            return ( MFAIL );
        }
        return ( ret );
    }
}

/* This routine initialises the bp - iop audit process.
arg :   none
returns : void
*/

void bpaud_init()

{
    own_qid_g = openqueue( own_key_g );
    addpid ( own_qid_g, BP_AUDIT_ID );
    selfid_g = gethostid ();

    /* get pathname from environment */
    path_g = getenv("GLBDATAP");

    /* Flush message queue */
    while( iop_receive(own_key_g, msgbuff_g, S_CBUFF,
IPC_NOWAIT) != MFAIL);
}

```

End of Code

Appendix D

References and Bibliography

- [1] K. B. Lal, T. Chandrasekaran, Y. K. Pandey "C-DOT DSS Architecture" - Overview of C-DOT & DSS Projects, October 1986.
- [2] K. B. Lal, T. Chandrasekaran, Y. K. Pandey "C-DOT DSS Hardware Architecture" - Overview of C-DOT & DSS Projects, October 1986.
- [3] B. Egert "Recovery Strategy For a Telephone Switching System" - Fifth International Conference on Software Engineering for Telecommunication Switching System, Conference Publication # 223, Southern Sweden.
- [4] Sunil S. Gaitonde, T. V. Ramabadran - "A Tutorial on CRC" - IEEE Micro, August 1988.
- [5] R. L. Engram, P. A. Shannon, S. S. Weber "Transparent Software And Hardware Changes In A Telecommunication System" - Fifth International Conference on Software Engineering for Telecommunication Switching Systems, Conference Publication # 223, Southern Sweden.

D. References and Bibliography

- [6] K. B. Lal et. al. "C-DOT DSS Software Arcitecture" - Overview of C-DOT & DSS Projects, October 1986.

- [7] C-DOT DSS TRAINING, LECTURE NOTES ON IOP & UNIX - C-DOT Official Publication.

- [8] C-DOT DSS TRAINING, LECTURE NOTES ON Software Architecture Overview - C-DOT Official Publication.

- [9] C-DOT 512 MAX General Description - C-DOT Official Publication.

- [10] C-DOT DSS TRAINING, LECTURE NOTES ON CENTRAL FILE SYSTEM - C-DOT Official Publication.

- [11] D. E. Knuth "Fundamental Algorithms: The Art of Programming Vol. 2" - Addison Wesley Publishing Company Inc. 1973.

- [12] M. Ben-Ari "Concurrent Programming Principals" - Prentice Hall International 1982.

- [13] Dennis M. Ritchie, Brian W. Kernighan "The 'C' Programming Language" - Prentice Hall of India, 1985.

- [14] Motorola Microsystems, Technical Documentation on System V/68 - # M68KUNPM/D1 Dec. 1985.

- [15] C-ISAM Reference Manual - Relational Database Management Systems Inc.