NAME OF AUTHOR *NOM DE L'AUTEUR*	Ken J. McDonell
TITLE OF THESIS *TITRE DE LA THÈSE*	A Unified Approach to Secondary Storage Input-Output Operations
UNIVERSITY *UNIVERSITÉ*	Alberta
DEGREE FOR WHICH THESIS WAS PRESENTED *GRADE POUR LEQUEL CETTE THÈSE FUT PRÉSENTÉE*	Ph D.
YEAR THIS DEGREE CONFERRED *ANNÉE D'OBTENTION DE CE GRADE*	1977
NAME OF SUPERVISOR *NOM DU DIRECTEUR DE THÈSE*	Dr. Tony Marsland

DATED *DATÉ* Sept 23, 1977 SIGNED/*SIGNÉ* [illegible]

PERMANENT ADDRESS/*RÉSIDENCE FIXE* [illegible] Leila Road
Carnegie, Victoria 3163
AUSTRALIA

THE UNIVERSITY OF ALBERTA

A UNIFIED APPROACH TO SECONDARY STORAGE INPUT-OUTPUT OPERATIONS

by

© KEN J. McDONELL

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE

OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA

FALL, 1977

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled "A Unified Approach to Secondary Storage Input-Ouput Operations", submitted by Ken J. McDonell in partial fulfilment of the requirements for the degree of Doctor of Philosophy.

T. A. Marsland
Supervisor

[illegible]

[illegible]

[illegible]

[illegible]
External Examiner

Date Sept. 13, 1977

ABSTRACT

Within present medium-to-large scale computer systems, the support of input-output operations involving secondary storage devices consumes a significant proportion of the resources and the operational budget. An initial survey indicates that current implementation strategies are far from optimal in two major respects; firstly, the maximum potential hardware resource utilization is not achieved, and secondly, the system software charged with supporting the secondary storage operations is poorly structured. As a consequence, reliability, system processing overheads, security enforcement, system adaptability and application program stability are typically deemed unsatisfactory for those tasks requiring, or providing, access to the secondary storage devices.

The uniform secondary storage input-output interface proposed in this thesis is designed to alleviate the shortcomings observed in conventional systems. Wherever practical, the interface development has followed an integrated hardware and software design philosophy.

Basically, the proposal hinges upon the mandatory imposition of a single software interface to the secondary storage resources for all operating system and user processes executing on the central processor. It is argued that the benefits of this approach are maximized if the interface is device independent and the permitted operations correspond to logical operations within a well-structured, conceptual data model.

Once the uniform software interface has been justified, further advantages are shown to result from moving the software which supports the interface out of the central processor, and into a dedicated external processor, having substantial independent processing capability.

The criteria applied throughout this research, when evaluating alternative design and implementation choices, are primarily qualitative in nature since the relative advantages and disadvantages often relate to factors which cannot be quantified. However, these qualitative evaluations are guided by the global objectives of reduced <u>total system</u> costs (i.e. purchase, maintenance and operation of both the hardware and software) and improved security enforcement.

ACKNOWLEDGEMENTS

I would like to thank Colleen McDonell for her patience, encouragement, and especially for the manner in which she has sacrificed her own ambitions and career goals, so that I could complete this work.

With respect to the content and its presentation I would like to thank Dr. Tony Marsland, my supervisor, for his encouragement and constructive criticisms. I am also grateful for the careful reading and comments provided by the members of my examination committee, Dr. Clement Yu, Dr. Bill Adams, Dr. Dale Bent of Computing Services, and Dr. David Hsiao of the Ohio State University.

I gratefully acknowledge the financial support of the Canadian Commonwealth Scholarship and Fellowship Committee in the form of a Commonwealth Scholarship, and the Department of Computing Science in the form of teaching assistantships.

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

Chapter Page

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

Chapter Page

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES (continued)

CHAPTER 1

INTRODUCTION

1.1 Identifying the Problem

Input-output operations involving secondary storage devices exert a significant influence upon the performance of present medium-to-large scale computer systems. For user programs, reliance upon these secondary storage resources varies from an obvious and direct requirement, as in the case of a large on-line inventory system, through to indirect dependence, for example, the apparently process-bound manipulation of large matrices in a virtual address space.

In typical installations, the resources dedicated to secondary storage operations extend beyond the physical hardware of the devices, controllers and data paths. Considerable amounts of processor time are devoted to setting up, verifying, initiating and checking the supported input-output operations, and a significant percentage of the software development and maintenance budget is appropriated to input-output related routines and subsystems.

One of the initial premises, which motivated this research, was the claim that conventional approaches to secondary storage input-output support have been found lacking in the following respects:

(1) The structure of the support software is generally poorly organized, and consequently the software development and maintenance costs are excessive in relation to the facilities which are provided, the enforcement of security and privacy constraints is inadequate, excessive central processor time is consumed in input-output related routines, and the management and utilization of the secondary storage resources is far from optimal.

(2) Rapid advances are already visible in the technology and speed of hardware components of the input-output subsystem, however the potential benefits of these changes are not being fully realized, as far as the system users are concerned.

(3) The range of supported input-output subsystems is needlessly large, due to the substantial overlap in functional capabilities between subsystems.

Consequently, the research was conceived as an attempt to verify these claims, based upon an investigation of current approaches to the provision of secondary storage input-output operations. If the postulated shortcomings can be established, then a second important goal is the formulation of proposals for an integrated hardware and software design to provide secondary storage input-output facilities for all user and operating system routines. It was hoped that such a unified approach would avoid some of the shortcomings of present systems, and permit full utilization of the emerging input-output hardware architectures with distributed processing capabilities.

1.2 Terminology

Before proceeding with the details of the research area and the approach, it seems essential to define some of the terminology which will be used.

Main Store:

The primary memory in the system, to which immediate access is provided for the execution of instructions by the central processor, and for the logical and/or arithmetic manipulation of data items. In addition to providing input-output buffer areas for the block transfer devices, main store is allocated to currently loaded (i.e. potentially executable) procedures, or parts thereof, and their associated local variables.

Central Processor:

The main computational module in the system, which executes instructions held in main store. This execution may be achieved by a variety of methods (e.g. hardwired control, microcode or some hybrid organization), and proceeds in one of two modes, 'supervisor' or 'restricted'. Supervisor mode is reserved for low level processes where <u>all</u> the resources of the machine are potentially accessible, however, some of the protection features of restricted mode may be selectively enabled. All other processes execute in restricted mode where, for example, 'memory protection' is enabled and certain instructions are designated 'non-executable'. Physically, the central processor may be either a single processor, or a homogeneous or heterogeneous collection of coupled processors.

Secondary Storage:

Encompasses all storage media external to main store and the central processor. Data and instructions held in secondary storage cannot be directly manipulated by the central processor, without prior transfer into main store. With current technologies, secondary storage typically consists of one or more direct access devices (e.g. fixed-head disk, moving-head disk, or drum), and is used as a main store paging area, for permanent data file storage, and for temporary storage (scratch files, spooling areas, etc.).

Input-Output Subsystem:

The hardware, firmware and software associated with secondary storage, but external to the central processor. This includes input-output processors, multiplexors, channels, device controllers and devices.

Operating System:

The group of software procedures which require supervisor mode processing; e.g. central processor scheduling and main store management (paging, protection, garbage collection, etc.). This concept of an operating system is similar to the 'nucleus' of the Multiprogramming System for the RC4000 machine (Brinch Hansen, 1970), where the primitive system functions are implemented in an environment which is not available to the processes which realize the higher level support functions.

Application Programs:

User supplied programs, system utility routines (compilers, editors, loaders, sort package, etc.), and some traditional operating system procedures which do not require supervisor mode processing.

Process:

A logically autonomous sequence of central processor instructions which is always executed without concurrency. A process may be an operating system procedure or a section of an application program.

Input-Output Module:

Responsible for the interface between a process and the secondary storage devices. Must co-ordinate both data f [illegible] tween main store and secondary storage, and co [illegible] / control functions between secondary storage a [illegible] entral processor. An input-output module is func [illegible] y [illegible] scribed by the set of operations it is capable of executing. Physically, this module may be implemented as a combination of software support routines which have traditionally executed as part of the operating system on the central processor, plus the hardware components of the input-output subsystem (e.g. channels, controllers, devices, etc.).

Secondary Storage Input-Output Operation:

Any operation initiated by a user, or a process which causes a secondary storage access to be initiated, or a secondary storage device buffer to be accessed, or invokes the process(es) responsible for management of the secondary storage resources. For the purposes of the current investigations, a secondary storage input-output operation consists of two parts, namely invocation (characterized by the interface between the requesting process and the input-output module), and execution (achieved by the input-output module).

Database Management System:

A collection of processes which support an integrated database.

1.3 Preliminary Assumptions Concerning the Operational Environment

Some of the more fundamental assumptions, regarding the environment in which a unified approach to secondary storage input-output support would be required, are discussed in the following paragraphs.

In broad terms, a typical configuration may suppo: one or more of the following computing environments:

(1) A time-shared operating system, suitable for on-line program development and execution on behalf of many concurrent users.

(2) One or more on-line database applications, providing a full range of transaction-oriented update, query and report facilities.

(3) A batch processing stream, providing less urgent access to the facilities of (1) and (2).

For such an installation, the program execution environment provided by a typical operating system would feature multiprogramming, virtual address spaces, spooled devices, and all the associated mechanisms for concurrent process interaction (e.g. resource allocation, protection, scheduling, and synchronization primitives).

To provide these operating system facilities, the base hardware would include at least one central processor, a large main store of the order of 10^6 bytes[1], secondary storage capacity in the 10^7 to 10^9 bytes range, a variety of local peripheral devices, and some telecommunications facilities (e.g. interfaces to processor-terminal and/or processor-processor networks).

1: 'Bytes' will be used as the unit of storage capacity throughout -- for the purposes of these discussions, a 'byte' and a 'character' are considered synonymous.

Implicit in the foregoing scenario is the assumption that large centralized computer installations will maintain their popularity and demands for service, despite attempts to introduce decentralized and personalized computing facilities. This assumption is based upon the the belief that while the smaller, decentralized processors will be used increasingly in those applications in which their cost-effectiveness can be demonstrated, many applications cannot be viably transferred to a smaller processor. Examples of applications or environments which are likely to maintain a heavy dependence upon a large centralized processor include, corporate data centers with very large databases, programs requiring fast execution in very large address spaces, universities and similar computer resource or service centers, programs requiring a variety of non-standard peripheral devices, programs in which the input-output workload is the most significant component of the reouired processing, satellite minicomputers with inadequate secondary storage, and where users require the full sp ::rum of services associated with large operating systems and their ancillary utility programs.

It should be noted that as programs and applications are relocated on decentralized user processors, the relative importanc f secondary storage input-output operations at the remaining centralized sites will, in all likelihood, <u>increase</u> because it is the jobs which are <u>not</u> heavily dependent upon the secondary storage resources which are the prime candidates for remote execution.

1.4 An Overview of the Research Directions, Methods and Conclusions

Chapters 2 and 3 present the historical overview and generalizations required to establish the inadequacies of present approaches to secondary storage input-output support. These problems are highlighted from both a hardware and software perspective.

As mentioned earlier, the balance of the research (i.e. Chapters 4, 5, and 6) has been directed towards using this survey material to help identify the desirable attributes for the software interface to an input-output module, and to specify a plausible structural basis for the module's implementation. As a direct consequence of these studies, <u>the following contentions form the central findings of the research</u>:

(1) For software executing on the central processor, the input-output module should present an interface which is homogeneous, independent of the physical devices in the input-output subsystem, and based upon logical operations within a conceptual data model.

(2) The software sections of the input-output module should reside in a dedicated processor having a substantial capability for independent processing with respect to

the central processor.

In the process of substantiating these contentions, the scope of the research has included the areas of database management technology, operating system principles, software engineering, performance evaluation, and macroscopic design techniques for multiprocessor architectures.

For the most part, the arguments, discussions and interpretations of alternative designs are based upon qualitative assessment of global constraints and those attributes of the system as a whole which may be deemed 'desirable'. From an early stage, the magnitude and scope of the planned research dictated that the current work should not be aimed at producing detailed systems designs. Rather, the objective throughout has been to establish a unified proposal, which appears feasible, and to identify the unresolved problems and areas requiring subsequent detailed investigation.

CHAPTER 2

THE EVOLUTION OF INPUT-OUTPUT SUBSYSTEM ARCHITECTURES

Input-output subsystem architectures have undergone significant evolutionary changes over the past 20-odd years. Early computers featured very limited input-output support. Transfers were executed one at a time, overlap between processing and input-output was not possible, all buffer storage was in main store, and all input-output related processing was performed on the central processor. As processors became faster with respect to the input-output devices, and hardware logic costs decreased <u>channel controllers</u> were introduced to permit multiple input-output transfers to proceed concurrently with central processor execution. Channel controllers also provided some rudimentary processing capability external to, and independent of, the central processor.

Further advances in hardware technology, coupled with a better understanding of multiple processor architectures has led to the development of <u>input-output processors</u>. These special function processors are generally minicomputer based, and capable of executing input-output operations which are both significantly more complex than those supported by channel controllers, and largely independent of the physical device characteristics. In addition, some of the resource management functions and housekeeping duties may be off-loaded from the central processor to an input-output processor. For database applications, the input-output processor approach has more recently been extended by the inclusion of special function processors, based upon non-numeric architectures.

Current technological and architectural predictions tend to favor the adoption, and extension of the input-output processor approach as the norm for medium to large scale machines in the next decade.

2.1 Early Configurations

Initial approaches to input-output implementation involved simple, device dependent interfaces to the central processor. The datum transferred across the interface was determined by the characteristics of the peripheral device. For example, attached to an IBM 1401 (IBM, 1963), the 1402 card reader-punch transferred a card image as 12 80-bit rows of information, while the 1403 line printer accepted zero to 100 characters in parallel, depending upon the contents of the output line and the print train position. These early input-output subsystems were characterized by:

(1) little or no buffer storage outside main store,

(2) very limited control logic and data manipulation

facilities outside the processor,

(3) no concurrency between multiple input-output transfers, even when associated with different devices, and

(4) complete dedication of main store and processor to input-output operations, since no overlap was possible between processing and input-output, except possibly for time-critical periods during the input-output cycle when the device was not engaged in data transfer.

Under these early machine configurations virtually all the processing associated with input-output operations was performed in software which was largely 'visible' to the programmer; e.g. record blocking and unblocking, character code conversion, buffer management, file label processing and format conversion.

2.2 Channel Controllers

By the mid 1960's the input-output architectures of most medium to large scale machines featured an arrangement which was radically different, compared to the earlier machines. Madnick and Donovan (1974, chapter 2) have attributed this change to the following facts:

(1) Compared to earlier machines, main store and central processor speeds had increased one thousand fold, while the peak transfer rates of peripheral devices had only improved by a factor of 10. The consequent reduction in central processor throughput during input-output

highlighted the inefficient use of the processor and main store.

(2) Substantial overlap of central processor operation and data transfers was technologically feasible using external logic which was specialized, simple, not too fast and cheap. These added logic modules were well suited to the necessary computation, manipulation and control functions, and provided a machine architecture which was economically attractive compared to the alternative designs based exclusively upon a general purpose central processor.

The resultant architectures featured five distinct functional units -- the <u>central processor</u>, <u>main store</u> and <u>peripheral devices</u> as in earlier machines, plus one or more <u>channel controllers</u> and a <u>main store arbiter</u> (refer to Figure 2.1). When coupled with an external interrupt mechanism and multiple direct data paths between the channel controllers and main store (i.e. by-passing the central processor), this architecture permits parallel execution of central processor instructions with not just one, but many input-output transfers.

The central processor is typically interfaced to one or more channel controllers, which are in turn interfaced to the peripheral devices. Physically, a channel controller may be constructed from two or more component modules, variously described elsewhere as input-output multiplexors,

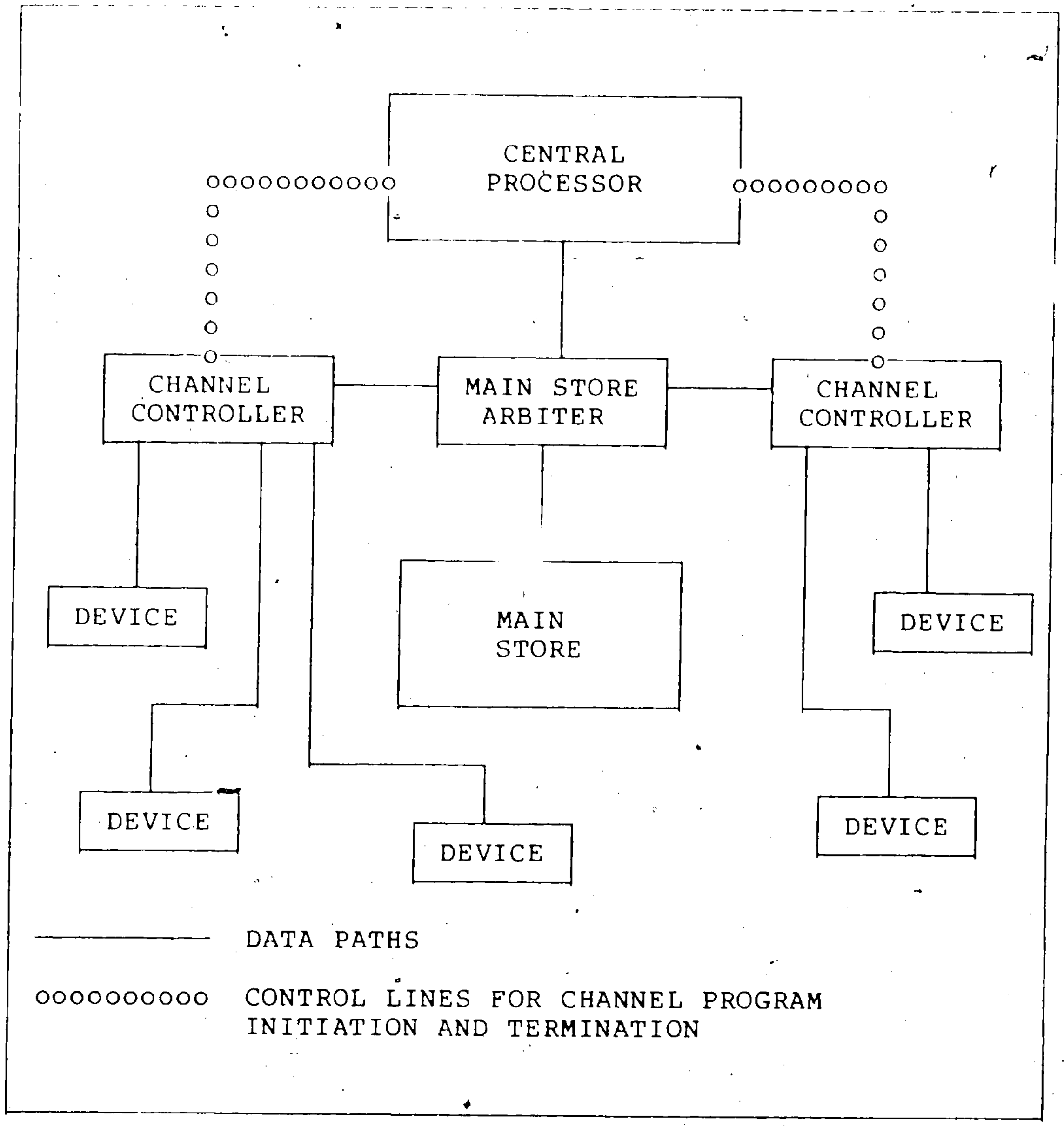

Figure 2.1. A Typical Channel Controller-Based Architecture

input-output processors,[1] data channels, device controllers, etc.

1: Refer to Section 2.4 for clarification of the difference between this type of channel controller and the class of special purpose processors described as 'input-output processors' elsewhere in this thesis.

Three areas of functional responsibility are assigned to a channel controller, namely channel program interpretation, device control and transfer co-ordination. By supporting the device-level functions (e.g. issuing device commands, servicing intermediate device interrupts, and multiplexing the main store data path(s) between the devices), a channel controller frees the central processor from the more mundane processing tasks associated with input-output transfers. In addition, the channel controller provides a single hardware interface between the central processor and multiple heterogeneous devices, thereby masking many of the device idiosyncracies from the central processor.

2.2.1 Channel Controller Functions

A process executing on the central processor initiates an input-output operation, or a sequence of operations by constructing a channel program from one or more channel commands. By way of an example, Figure 2.2 shows the semantics of the channel commands for a channel program designed to read one data record from an IBM 3330 series disk device (IBM, 1973a), assuming the required hardware resources are available, and the physical cylinder, track and sector address of the desired record is known in advance.

Once the channel program has been constructed and placed in main store, the channel controller is called upon to fetch and <u>interpret</u> the channel comands, one-by-one. Within the channel controller local scheduling strategies must be implemented to ensure that conflicting channel programs may be interpreted in a sequence which guarantees their individual integrity -- for example a program passed to the channel controller may request access to some physical resource which is already allocated for the interpretation of a previous channel program.

Interpretation of a channel command involves the second area of responsibility for the channel controller, namely <u>device control</u>. To achieve the intent of a channel command it is typically necessary to conduct a protracted 'conversation' with the device itself, to set and clear command lines, interrogate status, initiate pre-transfer operations, assemble characters, perform parity checks, hand device interrupts, etc. Device control functions are often delegated to a separate <u>device controller</u>, capable of servicing multiple homogeneous devices on behalf of a channel program interpreter.

Co-ordination of transfers between main store and multiple active peripheral devices constitutes the third function handled by the channel controller. Basically this involves multiplexing the data path(s) (i.e. the channels and subchannels) connecting the devices, device controllers, channel controller and main store, based upon the real-time urgency with which a given data transfer must be completed. The length of time for which a data path is assigned to a particular device is determined by the channel controller design, which in turn reflects the data transfer rates of the attached devices. For example, a slow device is typically assigned to a data path only during the transfer of a one or two characters (simple multiplexor mode), however a faster device remains connected to the data path while a complete physical record is transferred (block multiplexor mode), or for the duration of an entire channel program (selector mode).

```
      SEEK                   cylinder/track address
                             held in main store

LOOP: SEARCH ID EQUAL        ID (= cylinder/track/sector
                             address) held in main store

      BACK TO LOOP
        IF NOT EQUAL

      READ DATA              into main store buffer

Note: assumes the absolute cylinder, track and sector
      address of the record is known in advance --
      this may require some preliminary computation,
      a main store resident index or a previous channel
      program to access a disk resident index.
```

Figure 2.2 A Sample Channel Program to Read a Record

2.2.2 Main Store Accesses and Conflicts

In addition to requests originating from the central processor, main store accesses are required for the following purposes:

(1) The channel controllers must fetch the channel programs from main store, and return some status information to a main store location at the end of a transfer and/or a channel program.

(2) The input-ouput transfers involve main store accesses to fetch, or store, the transferred data -- an absolute main store buffer address is usually specified in the channel command which initiates the transfer.

In earlier machine architectures, all main store accesses were routed through the central processor, and the sequential mode of operation ensured that no main store contention was possible. However, the channel controller organization features multiple data paths to main store since there is at least one path for each processor and each channel, and possibly more in the case of 'multi-port' and 'multi-bank' memories. In theory, requests for memory access could be generated on all data paths simultaneously, or at least within the same memory cycle.

Concurrent main store accesses are prevented from interfering by the <u>main</u> <u>store</u> <u>arbiter</u>. In the event that two requests for main store cannot be simultaneously honored, the arbiter permits one access and delays the other, based upon the priority of the requests. Usually the priority scheme ensures that transfers involving the fast devices are serviced before requests from the slower peripheral equipment, which in turn are honored before accesses on behalf of the central processor

2.2.3 External Processing Capabilities

Besides freeing the central processor during input-output, channel controllers permit a small part of the processing associated with data transfers to be performed outside the central processor -- one of the most obvious examples being code conversion between a standard internal character set and various external character sets (e.g. program controlled EBCDIC to BCD translation performed by the controller on data passing between main store and a 7-track magnetic tape drive). However, some non-trivial features are also available, for example the 'File Scan' option on an IBM 2311 disk device/controller (IBM, 1969) supported the execution of simple character comparisons at the device (i.e. the SEARCH KEY AND DATA channel commands) -- this facility was subsequently enhanced and adopted as a standard feature for the later 3330 series devices.

Despite this external processing capability, the bulk of the input-output processing required to implement input-output operations at the level required by typical data management utilities (e.g. directory searching, logical record blocking and unblocking, and access via inter-record pointers) still has to be performed by software, executing on the central processor between the interpretation of channel programs. In fact, the one characteristic which distinguishes channel controllers from their more recent counterparts is the simplicity of the channel programs themselves. They are device dependent,[2] and very low level (e.g. read a block, rewind, seek, sense device status). Existing channel programs are so simple that their interpretation requires little external logic or processing capability -- to the extent that the cen [illegible] processor and a major part of the channel controllers for [illegible] achines (e.g. the IBM System/360 Model 50, (Flores, 969 chapter 7)) are both implemented via microcode on the [illegible] physical processor. As will be shown in Section 3.3.2, th[illegible] simplicity is achieved at the cost of considerable central processor overhead associated with preprocessing and postprocessing the channel programs.

In the light of their limited functional complexity, channel controllers appear to be rather expensive (i.e. not particularly cost/effective). Juliussen (1976) has shown that the cost of the controller may be many times the total

2: The _mechanism_ for constructing channel programs, initiating their interpretation and signalling channel program termination is _device independent_, and in fact this is one of the advantages of the approach. However the _channel commands_, and hence the channel programs, are singularly _device dependent_.

cost of the att[illegible]ed peripheral devices. It is expected that the integration of programmable microprocessors into the channel controller, instead of the current hardwired components, coupled with new integrated analog circuits and a shift in the manufacturers' pricing policies would reduce the relative costs of channel and device controllers.

2.3 Technological Factors

The desire to reduce the central processor overhead associated with input-output operations has led to a consistent trend away from the use of central processor resident software for the implementation of many input-output functions (Bachman, 1975; Withington, 1975). Alternative approaches include direct hardware implementation, firmware techniques, and software modules executing in processor(s) external to the central processor. Both low level and high level functions have been affected (e.g. 'bit picking' operations and logical record manipulation).

These changes have been made possible as a result of the design flexibility and economic advantages of processor components which have evolved following the advances in MSI and LSI technology. Consequently, there has been a wider acceptance of microprogrammable machines, distributed processor architectures and 'intelligent' device controllers (Barron and Glorioso, 1973; Berndt, 1974; Cooper, 1973;

Jensen, 1975; Lee, 1974; Rice, 1970). While the impact of this technological revolution is evident in all facets of computer architecture, the following examples serve to illustrate the range of potential applications concerned with input-output subsystems and input-output operations.

(1) Sindelar and Hoffman (1974) have described a secondary storage configuration in which an 8-bit microprocessor functions as a password security handler, and the disk controller implements a hardwired data enciphering alogorithm.

(2) A microprogrammable minicomputer has been used on an input-output channel to achieve autonomous data compression and expansion (Tao, 1974).

(3) The input-output architecture described by Poujoulat (1974) incorporates both hardware and firmware modules for the control of multiple disk devices. All scheduling and request queueing is performed within the input-output subsystem.

(4) For inter-record processing (e.g in searching and shifting operations), processor architectures have been extended, to include single microcoded instructions for descriptor-based 'character move', 'character compare' and 'hashing' procedures (Atkinson, 1974).

(5) Tomlin (1973) has proposed an intelligent, microprogrammable disk controller, capable of executing simple file management operations; for example, space allocation, retrieval of logical records from within a

phys block, linked list insertion and garbage collection.

(6) Acceptance of architectures incorporating input-output processors has depended upon the availablilty of processors with a low manufacturing cost, and high reliability (e.g. minicomputers). Examples of the use of the small, cheap, reliable processor technology within the input-output subsystem include the 'peripheral processor units' of the larger Control Data machines, which employ up to 15 programmable 12-bit processors each with a local 4K core memory and central processor connections (e.g. the CYBER 70 Model 76 (CDC, 1975b)), and Computer Automation's 'Distributed I/O System' (Computer Automation, 1976), which uses a hardwired 'I/O Distributor' along with microprogrammable 'Pico Processors', to support the central processor and device interfaces respectively.

Within the next decade, it is inevitable that new storage technologies will start to appear in commercially available product lines. The following scenario for likely secondary storage device attributes circa 1985 is constructed from published predictions (Baum and Hsiao, 1976; Hoagland, 1976; Martin, 1975; Withington, 1975) and the proceedings of a recent Symposium on Advance Memory Concepts (Miller and Gagliardi, 1976):

(1) Current fixed head disks will be replaced by magnetic bubble, electron beam or charge coupled (CCD) devices,

with a capacity of 10^8 - 10^{10} bits.

(2) High density moving head disks will retain their favored position for on-line bulk storage in the 10^{10} - 10^{12} bits range.

(3) Mass storage devices with capacities of 10^{13} and more will be based upon magnetic recording or holographic techniques. Unlike current magnetic tapes, these mass storage devices will require very little manual operator assistance.

2.4 Input-Output Processors

Within current computer systems, there are two commonly accepted approaches to input-output subsystem architecture. These two architectures are significantly different with respect to the distribution of processing capabilities between the central processor and the input-output subsystem. The 'IBM channel' is the archetype of the more centralized alternative discussed in the previous Section, while the distributed techniques cover a wide spectrum of architectures in which autonomous, special-purpose input-output processors provide access and manage the physical and/or logical data resources.

In the multiprogramming environment of a general-purpose computing facility there is considerable economic advantage associated with a system architecture which helps minimize the total time during which the central processor is either idle, interrupt processing, executing 'task swaps', or involved in mundane processing functions which could be handled by a smaller processor, like a channel controller. However, if a channel controller is to achieve an even greater autonomy and capacity for parallelism with respect to the central processor, then clearly the channel programs must be able to initiate more complex data transfer sequences and to achieve data manipulation without central processor intervention. This implies the following general modifications to convert a channel controller into an <u>input-output processor</u>:

(1) Upgrade the command interpretation and control logic in the channel controller to the status of a bona fide stored-program processor.

(2) Add considerable local storage for use as intermediate data and channel program buffers, and for holding the program(s) which interpret channel commands.

(3) Include device controllers with greater autonomy and a more sophisticated interface to the command interpreter (i.e. 'intelligent' device controllers).

An input-output processor accepts requests for <u>input-output operations</u> from the central processor. In general, a channel controller would have to interpret many channel commands, spanning multiple channel programs, to achieve a result comparable to the interpretation of one operation by an input-output processor. As an example, the input-output operation 'FIND RECORD X' could conceivably require a channel program considerably more complex than the one shown in Figure 2.2. Note that the channel commands, or their equivalent, are constructed from the requested input-output operation within the input-output processor rather than at the central processor as is the case for a channel controller organization.

The relationship between channel programs and input-output operations will be illustrated at greater length in Chapter 3 when the functional attributes of various software interfaces to the input-output subsystem will be discussed.

2.5 Frontend Communications Processors

Perhaps the most commonly accepted input-output processor is the 'frontend communications processor'. Within a time sharing or on-line environment, the frontend communications processor is charged with controlling all communication to and from the terminal devices, supporting line editing functions (e.g. backspace, character conversion, tabbing), data buffering and multiplexing the link between itself and the central processor. For example, the configuration described by Burner, Million, Rechard and Sobolewski (1969) uses an Interdata/3 as a dedicated processor servicing up to 32 active terminals, while dual PDP 11/45's at the University of Alberta are capable of supporting approximately 110 simultaneous terminal users, located both on-campus and at remote sites. With respect to both functional and hardware complexity, the frontend communications processor is clearly a significant extension of the classical channel controller architecture.

T input-output processor concept is not confined to a telecommunications environment. Similar processors are finding increasing acceptance as either general purpose 'satellite' processors (e.g. the 'peripheral processor units' on the Control Data Cyber 70 series machines, CDC (1971)), or as 'backend' processors between the central processor and the secondary storage devices. Indeed, the material presented in this thesis will concentrate upon the use of input-output processors in non-communications applications, to the virtual exclusion of frontend communications processors in the subsequent discussions.

2.6 Distributed Support Processors

External processors have been used to implement some of the input-output related functions provided by conventional operating systems. (A discussion of the analogous organization for supporting integrated database management systems will be presented in Section 2.7.)

Functionally, these distributed support processors provide services for a global operating system. In a typical application, integral sections of the operating system (e.g. the input-output control system, the file system, or the resource manager) are off-loaded from the central processor to a support processor. The support processors may operate either with considerable autonomous control, or under the supervision of some centralized, global control module, implemented in software or hardware. For heterogeneous configurations of support processors, the allocation of functions to particular components is normally static and designed to achieve the best possible match between the module's architecture and the function it must perform. If the system is configured with two or more homogeneous support processors, then the allocation of functions may be either dynamic or static.

For a Control Data Cyber 70 or 6000 Series machine running under the Kronos operating system (CDC, 1976) the supervisory functions are partitioned between a central processor resident monitor and a monitor executing in a dedicated peripheral processor. This second monitor assigns and releases blocks of main store, channels and devices on the basis of requests for service posted by both the central and the peripheral processors. Tasks are assigned to the 'free' peripheral processors by the decentralized monitor as the need arises. All peripheral processing units are identical, and capable of providing job scheduling, input-output control, job control language interpretation and system housekeeping services.

IBM's ASP subsystem (IBM, 1972) executes in one processor of a multiple-processor System 370 configuration.

ASP provides job control, spooling, operator assistance, inter-processor device sharing and media-to-media copying utilities for use by the other processors, running under the OS/VS2 operating system.

Management of storage hierarchies is another area in which distributed support processors may assist a central operating system. For example, the IBM 3850 Mass Storage System (IBM, 1975) uses 'on-board' microprocessors to handle the migration of data between the tape cartridge based 3830 mass storage device and a conventional 3330 series disk device, and to maintain the necessary data-set directories without any intervention on the part of the central processor based operating system. In fact, the mass storage device is not 'visible' from the central processor because externally the 3850 system supports a 3330-like hardware and channel program interface to the central processor.

Howie (1976) has suggested that a mass storage system comprising a mass storage device and multiple staging devices (e.g. moving head disks) has considerable potential as a replacement for conventional disk and reel-to-reel magnetic tape configurations. A self-managing storage hierarchy could provide economic storage with acceptable response times for a wide spectrum of applications, assuming some communication with the operating system to optimize performance. Practical applications would include an automatic bulk storage library, storage for system files, a general purpose file system, shared databases, distributed searching and subfile selection, and automated archival and backup procedures. The INFOPLEX system proposed by Madnick (1975) also features a self-managing storage hierarchy. However the INFOPLEX configuration is not based upon a mass storage device, but it does rely extensively upon distributed processors charged with localised optimization at each level in the hierarchy.

Record and file oriented operations may also be handled by external input-output processors. The MICS system (Ohmori, Koike, Nezu and Suzuki, 1974) incorporates a 'File Processor Module' - currently a minicomputer - which allocates disk space to the computational processor(s) and performs file management functions. All requests for disk input-output are handled by the File Processor.

Following a study of both the likely hardware developments and the historical evolution of file and database management systems, Gagliardi (1975) concluded that the functions of data management and storage management should be unified within a single dedicated processor. As a consequence, an architecture was proposed in which the 'storage subsystem processor' would be the centralized master processing unit, with all other processors (e.g. computational, spooling and communications processors) adopting subordinate roles. A recent paper (Bray, 1977) indicates that Univac are in the process of implementing a data management subsystem, based upon Gagliardi's model.

Hardgrave (1975) has proposed a computer system architecture in which multiple special purpose processors are connected via a communications network. Besides computational processors, the configuration featured special minicomputer nodes to support text editing and the communications subsystem, one peripheral processor per device for device control and network interface functions, and a 'set processor' providing a 'set theoretic database' interface to all the bulk secondary storage devices.

2.7 Processors for Database and Non-Numeric Applications

Advances in hardware technology have also eased the economic constraints on the development of viable unconventional computer architectures, specifically tailored for non-numeric processing, and in particular database applications. Goals in the design of non-numeric architectures include the reduction of input-output channel bandwidth requirements, easier software development and maintenance due to special instruction sets designed for non-numeric manipulations, reduction or elimination of time consuming index maintenance, increased parallel processing, response times which are independent of database size, and a closer match between the physical data storage structure and the user's conceptual information structure (Lipovski and Su, 1975; Ozkarahan, Schuster and Smith, 1975).

2.7.1 Searching Engines

It is becoming obvious that traditional, software-based searching procedures impose excessive demands on the central processor for rather simple character comparison operations, and cannot achieve the required response times demanded by those applications in which many on-line users are accessing a very large shared database. Alternative strategies have generally involved the replacement of the software procedures by special function hardware units, often supporting highly parallel modes of searching.

Since 1970, a great deal of investigation has been conducted into associative, or content-addressable, secondary storage units (Copeland, Lipovski and Su, 1973; Coulouris, Evans and Mitchell, 1971; Mitchell, 1976; Ozkarahan et al, 1975; Lin, Smith and Smith, 1976). Most of these proposals and implementations employ parallel-serial searching to emulate content-addressability on location-addressable devices (i.e. a serial search over many tracks or areas simultaneously). The necessary hardware modifications involve an alternate device controller or special logic within the device. Prototype devices in this class are quite powerful, being capable of many autonomous operations including, complex retrieval, update, deletion and garbage collection.

More recently, non-rotating storage devices have been used to build content-addressable memory modules with asynchronous search capabilities. The prototype version of the Relational Associative Processor (RAP) currently under construction at the University of Toronto is an example of this trend, where 'tracks' of disk storage have been replaced by CCD memory modules.

Conventional all-electronic associative memories are also being used to construct special purpose database search modules. Berra and Singhania (1976) have proposed the use of multiple associative memory units, based upon the capabilities of the STARAN architecture, to provide 'pipelined' searching of a hierarchic database directory or index. At maximum utilization, this architecture would allow simultaneous processing of N searches through an N-level tree-structured index. A more elaborate arrangement is provided by Honeywell's Extended Content Addressed Memory (ECAM) (Anderson and Kain, 1976). ECAM uses CCD technology to construct content-addressable arrays which operate under the supervision of a microprogrammable 'slave control unit'. Multiple slave control units are in turn controlled by a single 'master control unit' which also supports the central processor interface, query decoding (which involves the construction of microprograms for the slave control units), buffer management and maintenance of the storage structure descriptor tables.

The Leech processor (McGregor, Thomson and Dawson, 1976) has been developed for use in a database system based on the relational data model proposed by Codd (1970). This special function unit is capable of performing the relational operations 'join' and 'projection' as well as the conventional selection, sorting and merging functions, all under the general control of the central processor. Special modified searching algorithms have been proposed which exploit the Leech hardware and use a two pass 'coarse' and 'fine' search strategy to minimize the total number of records fetched and scanned on a character-by-character basis. When attached to a high speed bulk storage medium (e.g. a drum) it is anticipated that the searching throughput of the Leech processor will approach 100 megabytes per second.

The viable storage capacity of these database search engines is constrained by the cost of the add[illegible]l logic elements and the maximum density with which t[illegible]e logic elements may be integrated into the storage medium. It appears that the achievable capacity, which is in the order of 10^8 or 10^9 bits with multiple devices, will not be sufficient to permit the complete database to be loaded into the available search engines; therefore, some conventional location-addressable backing storage will be required. Under these circumstances, a search engine could be either dynamically loaded with sections of the database as dictated by the current search requirements, or permanently loaded with indices and database 'summary' records, to minimize the number of records fetched from backing store during the search procedure.

2.7.2 Backend Processors

The term 'backend' was intially coined to describe the experimental data management system (XDMS) constructed at Bell Telephone Laboratories (Canaday, Harrison, Ivie, Ryder and Wehr, 1974). Within XDMS, a considerable portion of the Univac DMS 1100 database management system was transferred from a Univac 1100 series central processor to a dedicated backend, in this case a Digital Scientific Meta-4 minicomputer. The backend processor was capable of executing commands expressed at the level of the CODASYL Data Manipulation Language (CODASYL, 1971).

Heacox, Cosloy and Cohen (1975) have described some preliminary, but promising experiments with a 'Dedicated Data Management' processor architecture modelled upon XDMS, while Lowenthal (1976) and Rosenthal (1977) have presented some of the rationale behind the backend approach currently being pursued by the MRI Systems Corporation for a new implementation of the System 2000 database management system.

The distributed database management system which resides on a network of heterogeneous minicomputers and is being developed by Kansas State University and the U.S. Army Computer Systems Command, features multiple backend processors. In this system, some of the network nodes also provide dual service, acting as backend database processors and host processors for the execution of application programs.

The relative merits and disadvantages of the backend organization will be discussed at some length in Chapter 5, however the use of a processor dedicated to the database management function seems to be justifiable in the following terms:

(1) The development of a dedicated database processor _is_ a viable proposition -- especially since current general purpose central processors are not _a priori_ the most cost/effective machines for specialized processing tasks (Flynn, 1977; Jensen, 1975; Juliussen and Bhandarkar, 1976). Provided that a reasonable match can be made between the throughput capabilities of the database processor and the central processor, then the total system performance will benefit from improved throughput and response time as a result of the increased parallelism.

(2) A database processor is a natural extension of the input-output processor organization, exhibiting increased functional autonomy, expanded local storage and more powerful processing capabilities.

(3) Software considerations favor a situation where the operating system can be developed and maintained independently with respect to the database management system and vice versa. This separation is greatly

simplified if the two software modules reside in separate processors.

(4) The central processor and main store resources allocated to the database management functions are significantly reduced.

(5) An <u>independent</u> database processor provides the potential for greatly increased system security and reliability. Security enhancements result from preventing direct access by central processor resident software to the secondary storage devices and not forcing the database management system to share a processor or, more importantly, main store with user programs. The interface protocol between the database and the central processsors could include automatic, bidirectional consistency checks, thereby greatly enhancing the capacity for early detection of processor malfunction and providing improved system reliability.

(6) Database sharing between multiple central processors in a homogeneous or heterogeneous configuration is readily supported. In a heterogeneous processor configuration, many of the data reformatting and translation problems vanish, since the physical data storage is controlled by the one database processor as opposed to multiple central processors. For programs executing in a network environment, it is much easier to establish communication between the central processor and a remote database processor than between the central processor

and a remote secondary storage device. For the secondary storage devices, software multiplexing via the database processor is a much simpler approach than the multi-processor interface and lock-out mechanism necessary for hardware multiplexing.

(7) The backend approach provides greater flexibility when a system upgrade is required, at less cost than an equivalent upgrade for a system with a central processor based database management system. For example, performance may be improved by adding an additional backend processor, or upgrading the existing backend processor(s), or keeping the backend processor(s) and upgrading the central processor.

2.7.3 Database Processors

Perhaps the greatest potential for the adoption of external processing capabilities in a database environment lies with the development of backend processors based upon novel, non-numeric architectures -- as opposed to software impl ntation on a conventional minicomputer. These specially constructed backend processors will be referred to as 'database processors'.

For example, a processor incorporating associative-type searching engines, a tailored instruction repertoire, hardware aided security mechanisms (suited to the protection of database objects in a shared, concurrent access environment), and firmware driven data structure translation could conceivably support a very efficient, inte ed database management kernel (Lowenthal, 1977). The availability of such a backend databas anagement kernel would ease the development costs for new applications, or alternative data models, whilst providing an acceptable throughput rate, a stable interface to the central processor, highly reliable operation and considerable capacity for local performance tuning and technological adaptation within the database processor and secondary storage subsystem.

Some designs for database processors have already been proposed. The original 'database computer' (DBC) design (Baum, 1975) included four specialized components within a single database processor. Three of these hardware components were based upon content addressable ar itectures and supported directory storage and manipulation, 'list' intersection and search evaluation, and the database storage modules. The fourth component enforced the security mechanisms, supported the central processor interface, performed command preprocessing and controlled the overall operation of the database processor. Subsequently, the DBC design has been extended (Hsiao and Kannan, 1977) to include 7 special purpose processor and storage modules.

Cook (1975) has described an ambitious project in which multiple 'user machines' would perform all the database input-output functions. Descriptions for user machines would be generated automatically from differences between the description of the logical and physical data structures. Each derived machine would then be emulated on a common microprogrammable host processor.

2.8 Projected Evolution and Development

Within the <u>database environment</u>, current implementation strategies and research proposals indicate that the database secondary storage devices will increasingly come under the <u>exclusive control</u> of a database processor (Whitney, 1973; Baum and Hsiao, 1976) -- the end result of this trend is a configuration in which the central processor is unaware of the secondary storage devices connected to the database processor. The database processor(s) will operate with considerable autonomy, simply communicating with the central processor at the beginning and end of a database operation.

Central processor resident programs will perform all database operations via the database processor, using requests phrased in terms of a <u>database interface language</u>. In its simplest form, a database interface language m permit individual logical records to be referenced using a unique identifier. Alternatively, the interface language may feature a rich query syntax to be used in defining a set of database records (e.g. the relational query languages, of which SEQUEL (Boyce and Chamberlin, 1973) is an example). Further consideration of the options related to the choice of a database interface language will be delayed until Sections 3.8 and 4.2.

Unfortunately, not all secondary storage devices are assigned to the database environment. As the discussions in Chapters 3 and 4 will show, functions such as paging, spooling and the provision of an on-line file system have historically been responsible for independently executing their own secondary storage input-output. This has resulted in a tight coupling between the physical device(s) and the central processor resident software. There is no evidence to suggest that the major manufacturers are planning any marked departure from this approach in the near future.

In a similar manner, it is to be expected that the unit record devices and mass storage devices will continue to be connected to the central processor via a conventional channel program or input-output processor interface. (The one possible exception being those mass storage devices dedicated to database archival and back-up procedures -- these devices may be attached directly to t database processor.) If the computer is to operate network environment, then a network interface processor will cc lete the repertoire of input-output related hardware pc nts.

Given the assumptions and postulates of this Section, medium to large scale computers which form the basis of the current investigations will generally conform to the hypothetical configuration shown in Figure 2.3.

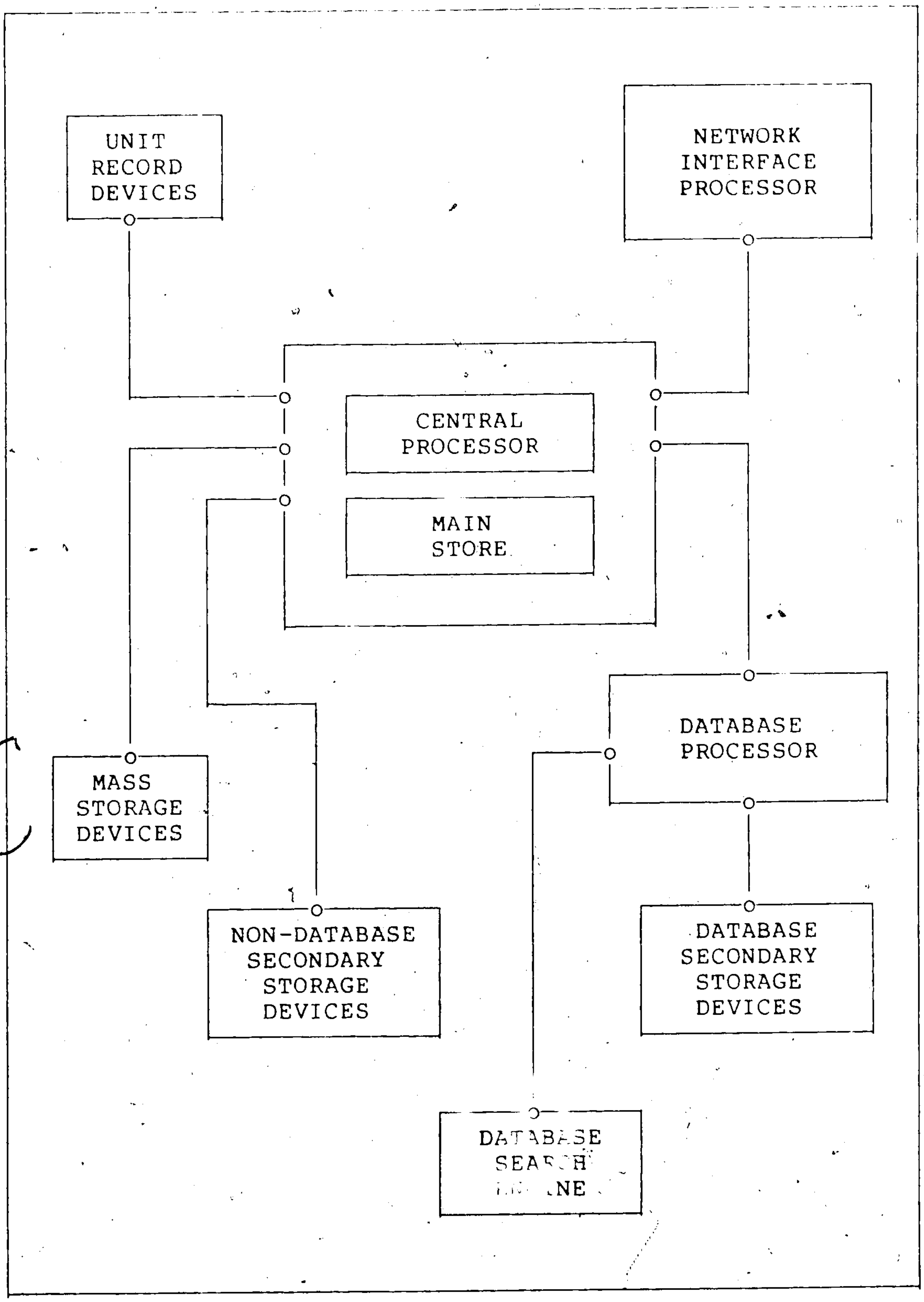

Figure 2.3 A Likely Computer System Architecture

CHAPTER 3

SOFTWARE INTERFACES TO THE INPUT-OUTPUT SUBSYSTEM

Within this Chapter the investigations will center upon the interfaces which are available to a software module when initiating an input-output operation from the central processor. The software modules under consideration include user written programs, utility programs, file systems, database management systems and operating systems. Emphasis will be placed upon those interfaces which are suited to operations involving secondary storage devices.

Input-output support for central processor based software, has followed an evolutionary path which has been both unstructured and highly incremental -- often following the input-output subsystem hardware developments discussed in Chapter 2. As improved input-output facilities became available, so new software support modules were developed. However continued support for existing programs dictated that the support modules be <u>appended</u> to the existing system, rather then <u>replacing</u> the superceded modules. Consequently, current systems typically provide <u>user</u> and <u>applications programs</u> with many input-output interfaces; e.g. channel program, stream data, spooled device, logical record, file system, standard file access methods, and database management system. In addition, the operating system generally supports multiple internal interfaces to the physical devices; e.g. the basic input-output routines within the paging, spooling, general purpose file and database management systems.

Clearly, a discussion of input-output interfaces implies studying the structure of, and the interaction between, the operating system, the input-output support subsystems, the input-output hardware and user programs. However, the input-output primitives of the available programming language(s) also impact the range and functional complexity of the interfaces available to a software module. Several 'applications oriented' languages will be analyzed with respect to the impact their input-output primitives have upon the potential interfaces.

Finally, processor network architectures, distributed databases and the operational constraints of a network environment will be examined to determine the possible influences upon the input-output interfaces.

The multiplicity of interfaces has led to a situation where the 'input-output module', as defined in Chapter 1, is not a unique unit. Rather, the specification of the input-output module for a particular software unit depends critically upon the unit's 'level' in the global system structure, the degree to which the operating system 'masks' or prevents access to posssible interfaces, and the programming language in which the unit was originally written.

Throughout this Chapter, the term 'interface' will be used to mean either the functional capabilities which may be invoked directly by a program, and/or the communication protocols and mechanisms associated with the execution of an input-output operation across the interface.

In order to bring some semblence of uniformity to the following discussion, the terminology and schematic data structure diagrams introduced by Bachman (1972) will be used to introduce the necessary terms and concepts. The terminology relates to the objects which may be manipulated, or accessed via an input-output interface. Bachman's terms and paraphrased concepts appear in Table 3.1, along with the new term 'File Catalog'. The relationship between these terms is illustrated by data structure diagrams which show the physical storage structure for a secondary storage device (Figure 3.1), the logical storage structure for a generalized file system (Figure 3.2) and a composite generalized storage structure (Figure 3.3).

TERM	CONCEPT
Block	Unit of storage allocation, and data transfer between the main store and secondary storage. Contains one or more physical records.
Cylinder	Unit of secondary storage, characterized by the fact that multiple accesses to one cylinder are much quicker than consecutive accesses to different cylinders.
Field	Smallest unit of data which may be associated with an attribute of some (real world) entity.
File Catalog	Central repository for the descriptions of all logical files.
Extent	Unit of storage allocation comprising a contiguously addressed portion of a volume. Accessed via its one or more component blocks.
Logical File	Named unit of logical storage which serves as a 'container' for logical records. Subdivided into pages.
Logical Record	Unit of logical storage accessed by a logical input-output operation. Contains zero, one or more fields.
Page	Unit of logical storage which has contiguous addressability when resident in main store. Contains one or more logical records.
Physical Record	Smallest unit of secondary storage which may be independently read/written.
Storage Device	The actual hardware unit, capable of holding a volume. Has a cylinder selection and read/write mechanism.
Track	Unit of secondary storage which provides the fastest serial transfer of consecutively recorded data. It is divided into physical records.
Volume	Unit of secondary storage capable of having data recorded upon it and subsequently reread. It is associated on a 1:1 basis with a storage device, and is divided into cylinders.

Table 3.1 Terms and Concepts Related to a Generalized External Storage Structure

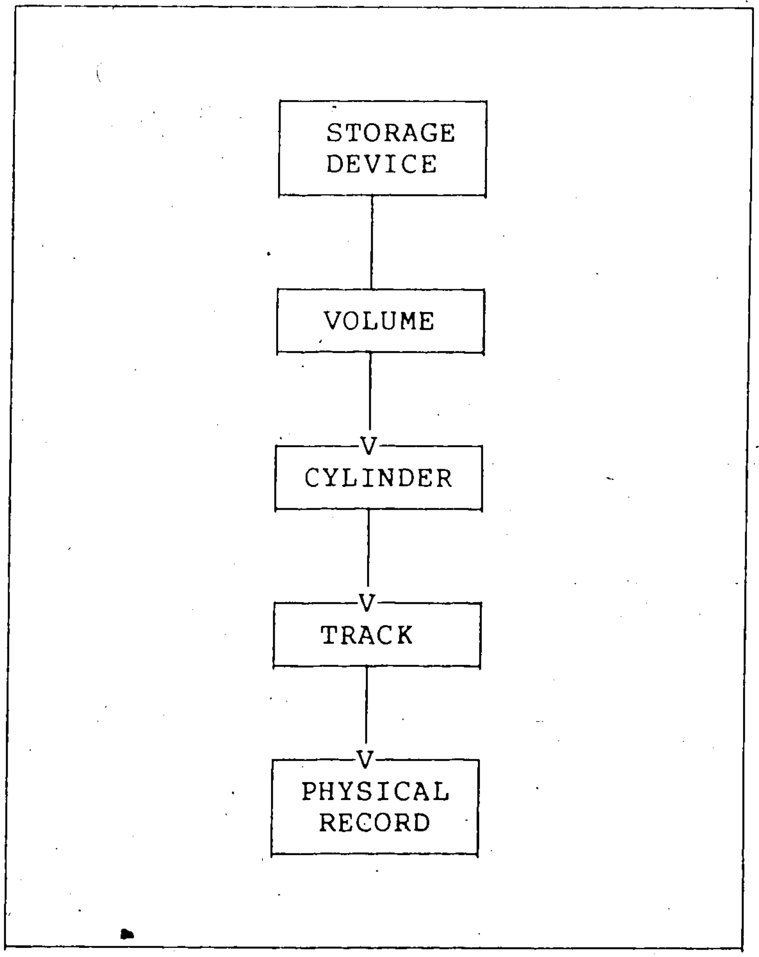

Figure 3.1 The Physical Structure of a Secondary Storage Device

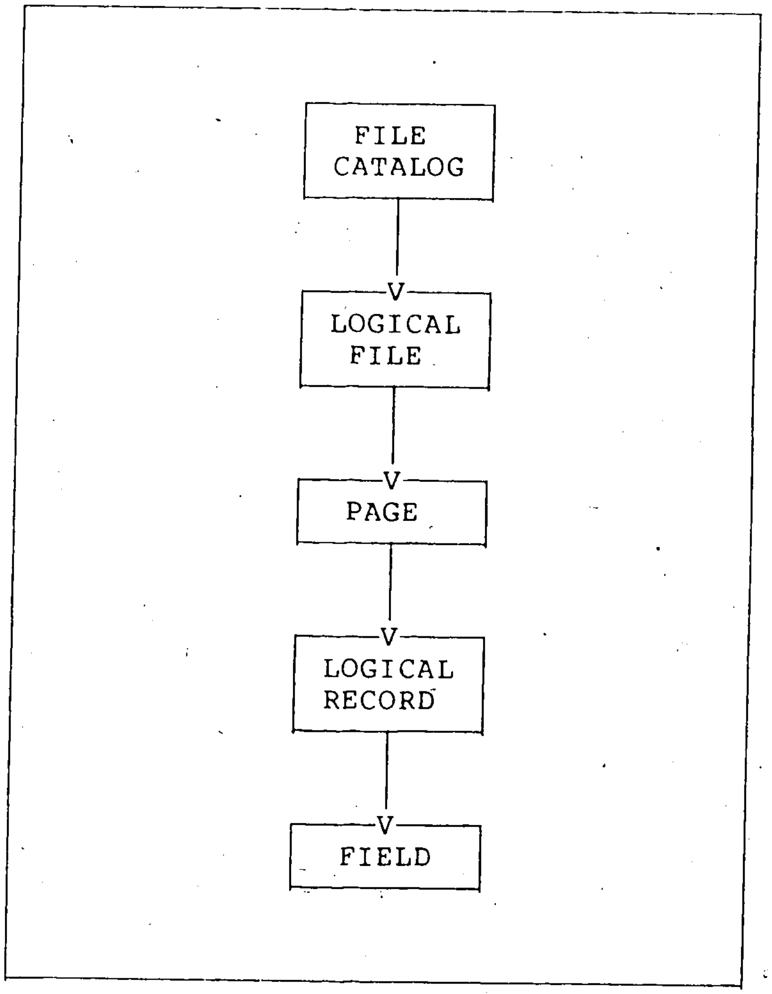

Figure 3.2 The Logical Structure of a File System

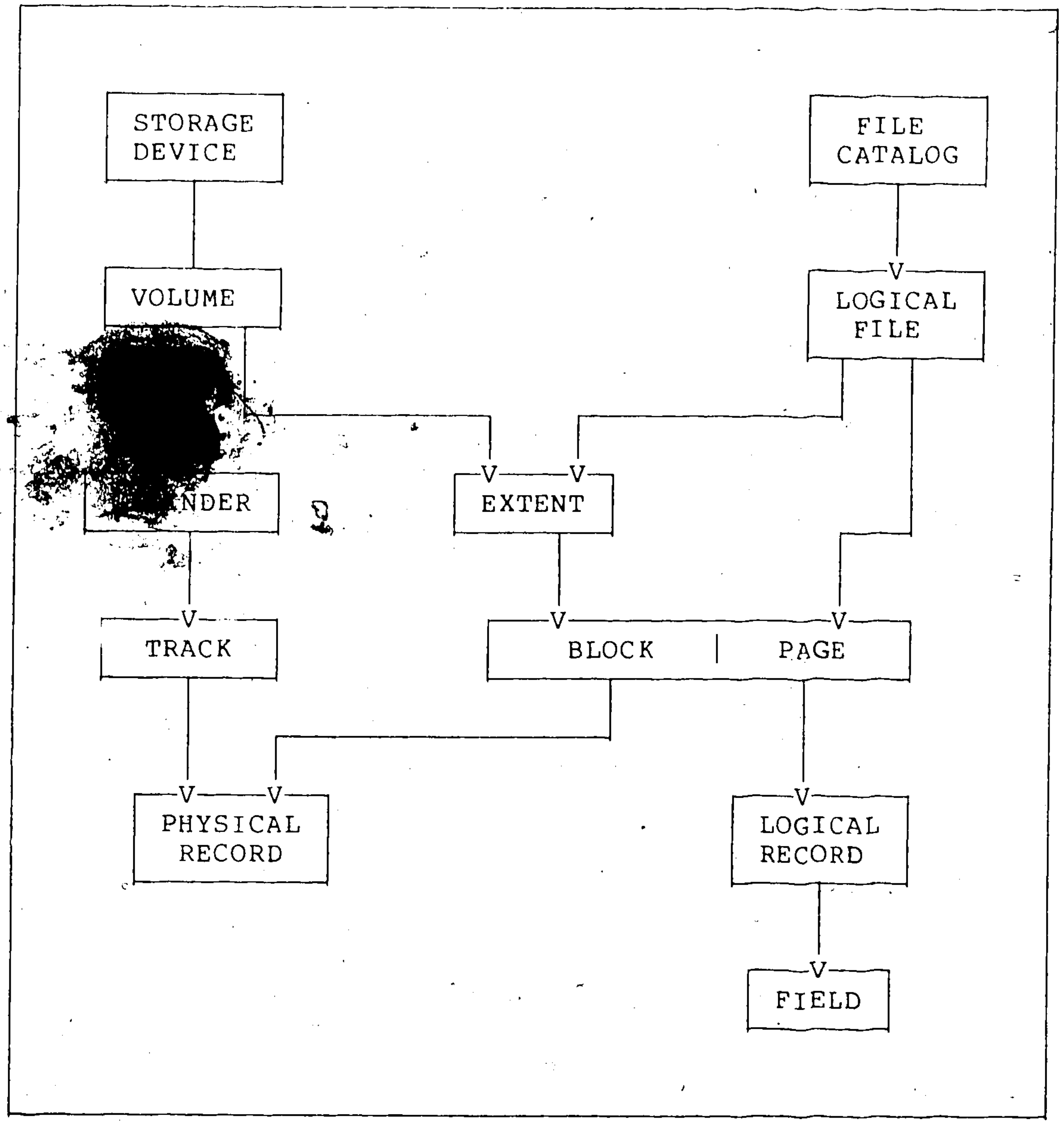

Figure 3.3 The Generalized Storage Structure

3.1 The Evolution of Multiple Interfaces

Early computer systems provided a single hardware interface to the input-output subsystem, and no software support. By comparison, current machines provide multiple hardware interfaces (with respect to at least functional capabilities, and sometimes invocation mechanisms), and multiple layers of support software implemented on top of the hardware interfaces. Together, these two attributes ensure a considerable variation in the range of input-output interfaces available to a software module.

In the following Sections, the development of multiple interfaces is discussed in the light of the hardware changes, the dynamics of input-output support routine evolution and the trend towards 'abstracting' input-output objects and operations.

3.1.1 Hardware and Architectural Influences

The most basic input-output interface is provided by the central processor's machine instruction repertoire. At this level, the central processor architecture and the hardware interface to the input-output subsystem effectively define the primitive <u>mechanisms</u> by which input-output operations may be executed. Table 3.2 illustrates the input-output related machine instructions for several medium-to-large scale central processors. Aspects of the input-output protocol which are visible to the machine language programmer and determined by the hardware design include:

(1) Dedication of fixed main store locations for status information, channel programs, pointers to channel programs, the device and channel controller description tables, and other 'housekeeping' data areas.

(2) Limits upon the device / channel controller topology and addressing conventions.

(3) Procedures for interrupt handling.

(4) Formats for channel commands and the status information.

Techniques are often provided to ensure that routines outside the central operating system do not have direct access to the machine's input-output primitives. Rather, hardware enforced selective execution (depending upon the 'processor state') is coupled with a shared, software service routine and a software interrupt mechanism to give indirect, selective and controlled access to the machine's input-output instructions. For example, the 'UUO Handler' within the resident portion of the DECSystem-10's operating system (DEC, 1975), and the 'Supervisor Call' (SVC) mechanism for IBM's 370 series (IBM, 1974) both execute primitive input-output operations as a service function on behalf of a calling process. A further discussion of the attempts to 'mask' or 'hide' some of the input-output interfaces from some of the executing processes will be presented in Section 3.7.

Machine	Input-Output Instructions	Instruction Operands
IBM Series 370	Start I/O	channel & device address, channel program address.
	Start I/O Fast Release	same as for Start I/O
	Test Channel	channel address
	Test I/O	channel & device address, resultant status
	Clear I/O	same as for Test I/O
	Halt I/O	same as for Test I/O
	Halt Device	same as for Test I/O
Burroughs B6700	Scan In	input-output processor address, function descriptor and operand.
	Scan Out	same as for Scan In
Honeywell Series 60 Level 66	Connect I/O Channel	channel program address
CDC 6000	Central Exchange Jump	initial register values for a peripheral processor program
DEC PDP 10	Conditions In	device and control word address
	Conditions Out	same as for Conditions In
	Data In	device and buffer address
	Data Out	same as for Data In
	Block In	device address, buffer address and count
	Block Out	same as for Block In

Sources: IBM (1974), Burroughs (1972), Honeywell (1975), CDC (1975a), and DEC (1970).

Table 3.2 Some Typical Central Processor Input-Output Instructions

Alternative approaches to the conventional 'interrupt based' communication between the central processor and the input-output subsystem have been implemented using hardware and firmware techniques. The Venus machine (Liskov, 1972) incorporates a microprogrammed Input-Output Channel which communicates with a process executing on the central processor via a hardware extension of the classical semaphore mechanism (Dijkstra, 1968), rather than via asynchronous interrupts. Besides a standard interrupt mechanism, the Burroughs machines (e.g. the B6700) use hardware assisted queue maintenance operators to construct lists of pending and completed input-output operations (Burroughs, 1976; Patel, [illegible]69). These lists are manipulated by intrinsic system functions, which may be invoked by the requesting process executing in a central processor, or by other intrinsic functions, or by the input-output processor(s). Outside the nucleus of the the Multics system (Organick, 1972), interrupts are not visible and asynchronous interprocess communication is supported with four semaphore-like primitive operations. In this way, the user and operating system processes may synchronize themselves by temporarily suspending execution, pending some short-term system event (e.g the arrival of a page in mainstore), or pending some long-term process event (e.g. a co-operating process notifies the suspended process that it may now continue execution).

The architecture of the input-output subsystem exerts a strong influence upon the functional characteristics of the basic input-output interface available to a process executing on the central processor. Conceptually, the potential interfaces fall into two classes, based upon the complexity of the operations which may be performed across the basic interface -- the functional complexity of the interface in turn reflects the extent to which the input-output subsystem architecture can support autonomous processing functions. The functionally simpler category features a physical record interface (e.g. as supported by a channel controller organization), while the more complex group includes the logical file and logical record interfaces (the direct implementation of which requires an input-output processor organization[1]). One important distinction between a logical and a physical interface centers upon the translation of a record's address from a logical identifer to physical storage address. Operations across physical interface must include a physical address as an operand, therefore the translation must be done in advance by central processor resident software. For a logical interface, requests are independent of the information's physical location, and consequently the translation is performed outside the central processor. The attributes of these two fundamentally different input-output

1: As will be shown in Section 3.4, the logical interfaces may also be implemented indirectly, using one or more layers of central processor based software on top of a physical input-output interface.

interfaces will be discussed in Sections 3.3 and 3.4.

3.1.2 Input-Output Support: The 'Add-On' Growth Phenomenon

The introduction of software based input-output support packages as extensions of the basic central processor input-output facilities was motivated principally by a desire to make the programming of frequently encountered input-output operations easier. Subsequent considerations included security and integrity threats, and poor resource utilization, which were overcome to some extent by providing input-output control services which give the programmer indirect, rather than direct access to the basic input-output facilities.

Input-output support routines have generally evolved parallel with the development of new architectures or improved performance within the hardware components of the input-output subsystem. First came the 'Input-Output Control Systems', which freed the programmer from some of the more mundane and tedious aspects of input-output operations associated with sequential file processing. The implemented control functions were naturally influenced by the characteristics of the magnetic tape storage medium (e.g. sequential blocked access) and the dedicated nature of the peripheral unit (e.g. no concurrent shared access to a file or device).

Introduction of direct access storage devices (e.g. disks) led to the upgrading of the input-output support to provide a class of generalized 'File Access Methods' and database management systems. As with the earlier input-output control systems, the device attributes influenced the range of operations supported by the file access routines (e.g. direct and indexed sequential access methods, and device sharing between active processes). However, the operations and access methods implemented under an integrated database management system are device independent -- they are specifically tailored for concurrent access to a shared file or database. A typical database management system supports a range of capabilities which include, as a subset, all the operations supported by the input-output control and file access routines.

Implicit input-output support was introduced with the advent of virtual memory and multiprogramming systems via the paging and/or swapping routines. For network environments, special software modules were developed to support the hardware interface to the network, the 'line' protocols and the 'host-to-host' protocols.

While these support systems are functionally related (i.e. they make input-output easier, at some conceptual level above the machine's basic input-output instruction repertoire), they are often constructed as physically disjoint modules, each implemented on top of its own low level routines for achieving physical input-output (e.g. channel program construction and execution). For example, a current computer system may provide input-output support routines for sequential devices, spooling, paging, the file system, file access methods and a database management system. Routines which are shared between these subsystems have typically evolved in an ad hoc manner, compounded by the peculiarities and inadequacies of the precursor subsystems.

In addition to the unstructured implementation of these various support subsystems, the functional capabilities of the software interfaces have considerable overlap (e.g. there are many ways in which a program may execute one input operation). In part, this redundancy may be attributed to an unwillingness on the part of the suppliers to abandon earlier support systems. Clearly, the unilateral adoption of a new support system would have precipitated considerable software modification within existing programs. Another motivation for maintaining multiple input-output support subsystems is that, despite the considerable functional overlap, subtle differences remain. No attempt appears to have been made to separate the common functions, which could be implemented in a central support module, from the non-standard functions requiring separate, specialized routines.

To further confuse the issue, each input-output interface supports its own communication protocol. Leaving aside the semantics of the operations executed via the multiple interfaces, the variation in intermodule communication mechanisms may be illustrated by the following example; the range of module linkage conventions for OS/360 (IBM, 1971) includes 4 parameter passing protocols and 6 techniques for the transfer of control -- for a grand total of twenty-four 'standard' (!?) interfaces.

The problems associated with multiple input-output support subsystems will be dealt with at greater length in Chapters 4 and 5, however at this stage it should be noted that the evolutionary approach has resulted in a software structure which is often difficult to comprehend, unnecessarily large, and prone to security violations.

3.2 Multiple Classes of Input-Output Abstraction

The repertoire of input-output operations presented to the applications programmer may be influenced by various 'abstractions' of the input-output devices, external data structures and input-output operations.

The technique of abstracting or virtualizing objects in the input-output environment exhibits considerable similarity to the concept of an 'abstract data type' found in some programming languages (Gerschke and Mitchell, 1975; Liskov and Zilles, 1974). An abstract data type is implemented as a set of user callable operations which provide no information concerning the internal procedures or data structures used to implement the operations; e.g. a 'stack' may be defined by the operators 'push' and 'pop', and the implementation details are irrelevant for a procedure wishing to use a 'stack' data type. The benefits of this approach include improved program structure and reliability, easier program modification and simplified proofs of program correctness.

The use of abstract input-output objects has been most evident in the design and development of multiprogramming systems to be used as vehicles for operating systems research. When implemented, these systems typically run on rather small hardware configurations, however the concepts seem to be equally applicable to larger systems.

Two basic techniques have been applied, namely the 'layered' or 'hierarchic virtual machine' approach, and the 'concurrent co-operating process' or 'monitor' organization. Within the former category, the THE system (Dijkstra, 1968) is the archetype, in which successive layers (or virtual machines) support abstractions of the processor, main store, operator's console and peripheral devices. Routines communicating with a <u>virtual</u> <u>device</u> are unaware of physical device allocation, scheduling, buffer management, physical transfers or interrupts. Gagliardi (1975) proposed three further levels of abstraction, in an attempt to upgrade the complexity of the input-output functions supported by Dijkstra's original model. These new layers provided logical record input-output, file access methods and database management services.

A user process executing in the environment provided by the Venus machine (Liskov, 1972) may communicate with 'virtual devices' which are supported by the microprogrammed operating system nucleus and three levels of software. In attaining the visible attributes of a virtual device, the software and firmware routines incrementally 'mask off' the mechanics and real time constraints of device communications, the buffering procedures, the queueing and scheduling of outstanding requests and the necessary interprocess synchronization.

The RC 4000 system nucleus (Brinch Hansen, 1970, 1971) provides an environment in which all input-output objects are uniformly treated as external processes. Internal processes communicate with themselves and exter l processes by means of a message exchange mechanism.[2] One group of the RC 4000 'monitor functions' support operations involving abstract objects (i.e. external processes) corresponding to

2: Blasgen (1975) has pointed out that using an interprocess communication facility to execute input-output asynchronously, with respect to the execution of the initiating process, poses some serious disadvantages in certain applications. Specifically, for terminal input-output, real-time control devices, and handling input-output errors, a more efficient but equally elegant solution would involve the use of synchronous input-output primitives which suspended executing of the calling process until the input-output operation was completed.

the non-secondary storage devices and logical files which are named, contiguous blocks of secondary storage. Multiple operating systems may be implemented on the single RC 4000 nucleus. Boss/2 (Lauesen, 1975) is one such operating system which provides a further abstraction in the form of virtual devices, implemented via a spooling mechanism. Despite the outward symmetry between external and internal processes, significant differences were evident with respect to processor scheduling and access to the nucleus's data structures. Brinch Hansen (1973a) subsequently identified this distinction as one of the artificial constraints and disadvantages of the RC 4000 nucleus.

The concept of a 'monitor' as a basic component from which an operating system could be constructed has evolved from studies of various semaphore and critical region techniques. Monitors were first described by Dijkstra (1971) and subsequently formalized by Hoare (1973, 1974) and Brinch Hansen (1973b). Because monitors are based upon the notion of mutually exclusive execution of shared routines and mutually exclusive access to sharable data structures, they are well suited to handling the types of asynchronous, conflicting requests for input-output service found in a multiprogramming system.

More recently, the monitor concept has been included as an integral part of the Concurrent Pascal programming language (Brinch Hansen, 1975). Concurrent Pascal has been used to implement the Solo operating system (Brinch Hansen, 1976a, 1976b), which supports abstractions of the physical devices, disk files (i.e. sets of contiguous physical records), and logical files.

3.3 Physical Interfaces

As mentioned earlier, the spectrum of potential input-output interfaces may be partitioned into two generic groups, namely the logical interfaces and the physical interfaces. In this section, the common attributes of the physical interfaces will be described, with particular attention being paid to the channel program interface.

A physical interface is characterized by the presence of a <u>physical record address</u> as a parameter in all transfer operations. In general terms, a physical record address comprises a <u>device address</u> (usually a channel and device number pair, or a unique name associated with one of a class of identical devices), plus an <u>absolute storage address</u>. For strictly sequential devices (e.g. card readers, line printers, or paper tape equipment) the absolute storage address is al[illegible]s 'the next physical record', for block addressable devices (like disk or drum units) it is a physical record number relative to the start of the volume or storage medium.[3]

Given a physical input-output interface, then central processor based software outside the input-output module must be used to implement both the secondary storage space allocation and the control of concurrent access to shared devices[4] and/or secondary storage areas.

The units of data transferred across a physical interface are either blocks or physical records, formatted according to the predefined external storage layout -- often the user's buffer areas are also used as device buffers. Consequently, the input-output module is not responsible for any format conversion (other than a possible character-by-character translation between internal and external character codes), subrecord selection or dynamic (e.g. table driven) reformatting. In short the input-output module treats the transferred data as integral unit which is independent of all other physical records and not subject to any semantic interpretation.

3: Note, not the start of a logical file or relocatable extent; for a disk device, the absolute storage address would be composed of a cylinder number, a track number within the cylinder, and a physical record number within the track.

4: Queueing the pending input-output requests to ensure exclusive access to a device for the duration of a single operation or channel program would typically be done within the input-output module. However, any integrity based constraints spanning multiple input-output requests from a single process, or high level (e.g. transaction oriented) scheduling must be enforced outside the input-output module.

Some physical interfaces have already been discussed in this Chapter; for example, input-output related machine instructions, early input-output control systems, and the abstract devices which have a 1:1 correspondence with an actual peripheral unit. However the material covered in the following Sections will concentrate upon the most widely encountered physical input-output interface -- the channel program interface. The discussion will cover the channel program execution mechanisms, the constraints upon the functional complexity of channel programs, and the observed advantages and disadvantages of the channel program interface.

3.3.1 Initial Advantages

The initial advantages of a channel program interface were discussed in Section 2.2, and the points raised included:

(1) Increased parallelism and concurrent processing for improved throughput performance.

(2) The use of cheap external logic yielded superior cost/performance ratios.

(3) The lowest level device control functions were divorced from the central processor.

(4) A device independent mechanism was introduced for the execution of input-output operations.

Despite these advantages, the following discussion will show that the channel program interface has some serious shortcomings for current input-output environments.

3.3.2 Processing Overhead

In his survey of input-output subsystem architectures, Buzen (1975) identifies some of the factors which have promoted and hampered the development of channel program interfaces. If parallel execution of input-output operations and central processor instructions is permitted, then some synchronizing mechanism has to be adopted -- the universally accepted approach has been the 'I/O Completion Interrupt'. But interrupt processing imposes a non-trivial overhead, associated with saving the status of the interrupted task and initializing the interrupt handler. Consequently, large block transfers and multi-command channel programs (i.e. data and command chaining) have been adopted in an attempt to reduce the total number of interrupts.

Some machine architectures exhibit features which reduce the overheads associated with interrupt processing and changing processor state. The first technique involves a 'stack' rather than a 'general purpose register' organization, since the current state of the system and the interrupted process remains in the stack and does not have to be copied to a static 'save area'. Examples include the larger Burroughs machines and the ICL 2900 series (Doran, 1975). A second approach involves the use of multiple sets of general purpose registers (e.g. one set per processor state, or one set per interrupt level) -- again the advantage is that no explicit saving of state information is required when the processor switches task or [illegible]ion mode. Examples of this second approach include the In[illegible]data 8/32, the PDP 11/45 and the ModComp IV.

For virtual memory systems, considerable preprocessing is required before a channel program can be forwarded to a channel controller for execution. Typically, a 'page fault' in the middle of a disk transfer constitutes an unrecoverable error, short of restarting the channel program. Therefore, it is necessary tc ensure that all <u>virtual pages</u> which will be accessed as a result of executing the channel program are brought into main store and flagged as 'unpageable' (i.e. locked in main store). This must be done before any input-output transfer is intiated, and involves a software routine which preprocesses the complete channel program.

Since the channel controller does not have access to the 'page tables', further processing overhead results from translating all virtual memory addresses into the real, main store address space. This must be done for all buffer and operand addresses cited in the channel program. At the same time, the validity of the requested operation and the supplied addresses for buffers, physical storage locations and operands must be checked against the access privileges of the requesting process. Once the parameters have been checked and translated, the channel program dispatching routines must ensure that the parameter values are not subsequently modified prior to their use by the channel controller. For systems which permit self modifying channel programs, unconstrained transfers of contol within a channel program, or dynamic channel program construction, the preprocessing procedures become very complex, error-prone and time-consuming

Some novel and some messy techniques have been developed to reduce the over d associated with channel program preprocessing. Whi Welch (1975) have described a modular main stor design in which the ' t to real" address translation tables re integrated into the storage modules. All main store accesses are in terms of virtual addresses, and the main store hardware handles the translation to real addresses and the necessary page allocation and replacement algorithms. Besides avoiding channel program address trans ation, this scheme provides faster central processor state changes, since there is no cache me ry to be 'flushed' and no page table control registers to be set up.

A similar philosophy is involved in the 'intelligent paging device' which Wayne State University is currently considering as a viable replacement for a conventional paging drum.[5] This device would handle its own space management, and provide access to virtual pages on the basis of a process identifier and a virtual page number.

The Michigan Terminal System (MTS) avoids the address translation phase for all paging operations, by having the paging routine execute in the real address space (McDonell and Marsland, 1977).

3.3.3 Security Considerations

One of the most critical problems associated with the channel program interface is its use as a mechanism for penetrating the operating system and violating the system's security constraints.

Linde (1975) has identified three generic weaknesses associated with low level physical input-output interfaces which are common to many operating systems:

(1) Self modifying channel programs and/or dynamic channel program construction provide mechanisms whereby the validity checks associated with main store accesses may be bypassed, once channel program execution has commenced.

(2) Channel controllers typically have unlimited asynchronous access to mainstore. As a consequence, little or no access privilege checking is performed <u>at</u>

5: See the <u>SHARE MTS Newsletter</u> no. 40, June 1977.

the time the input-output transfer takes place.

(3) The operation of the system may be halted or drastically downgraded by a channel program which assumes control of an input-output path, and does not release it (e.g. a channel program containing an infinite loop).

Virtual machine architectures have been widely proposed as one solution to the problem of enforcing protection protocols and constraints upon concurrent users who require access to all the machines's hardware facilities (Madnick and Donovan, 197[illegible], Popek and Kline, 1974). However, an organized attempt to penetrate the VM/370 virtual machine system (Attanasiao, Markstein and Phillips, 1976) revealed that, "almost every demonstrated flaw in the system was found to involve the input/output (I/O) facility in some manner". These weaknesses were attributed to VM/370's attempts to construct real channel programs from virtual channel programs, duplication of input-output services within the VM/370 monitor and parallel execution of virtual machines and channel programs.

It seems highly unlikely that the security requirements of the next decade will be met by any computer system featuring a physical input-output interface which is as low level, generally accessible, and uncontrolled as the current channel program interfaces.

3.3.4 Database Access Patterns and Requirements

Execution of a single database-level input-output function involves the use of multiple, non-trivial channel programs. Constructing complex channel programs at the central processor involves considerable overhead which either may not be necessary, or would be less expensive if the channel programs were constructed in an external input-output processor.

The repertoire of channel commands which are available for disk devices are not particularly well-suited to database operations. For example, searching at the device is limited to one key area within a fixed physical record format, and for short records (e.g. index records), device level searching may be <u>less efficient</u> than transferring a block of records to main store and then using central processor based software to search the block (Buzen, 1975).

If the input-output operations required by a program involve physical-to-logical record restructuring, subrecord selection, index accessing or stepping through 'linked' storage structures then a good deal of extraneous data is passed between the channel controller and main store. In fact, multiple physical records must be transferred to main store before a logical record can be constructed by central processor resident software -- this means an unnecessarily high bandwidth data path and more work for both the central processor and the main store arbiter.

3.3.5 The Demise of the Channel Program Interface?

In addition to the disadvantages outlined in the preceding Sections, the channel program interface is simply not required by user processes, or by most operating system procedures. Historically, low level device control and a physical record interface were provided in response to:

(1) a monoprogramming environment in which individual programs exploited their control over dedicated devices to reduce run time by overlapping input-output with processing, and

(2) machine architectures in which central processor intervention was required to perform input-output related processing.

These conditions have changed with the introduction of multiprogramming systems and input-output processors with substantial independent processing capabilities. Multiprogramming has meant that an individual program no longer has guaranteed, dedicated control over a device and the responsibility for achieving overlap of operations has been assumed by the operating system. Concurrency between input-output transfers and central processor execution is typically achieved by overlapping the central processor execution of one job with the input-output transfers of a second job, rather than forcing the one monoprogrammed job to simultaneously execute instructions and external data transfers.

As input-output subsystem architectures incorporate further external processing capabilities, and security considerations assume increased importance, it is to be expected that the importance of the channel program interface will undergo a rather critical revaluation. Over the long term, it seems unlikely that such a functionally simple interface will survive, with the possible exception being for some very low level modules of the operating system, charged with servicing non-standard or real-time devices. Further justification for the demise of the channel program interface will be presented in Sections 4.1 and 4.2, when the potential advantages of a homogeneous input-output interface are discussed.

3.4 Logical Interfaces

A logical input-output interface is characterized by the absence of the physical record address which is mandatory for all transfers across a physical interface. The unit of data passed across a logical interface is a <u>logical</u> <u>record</u>.

Within the input-output module, the logical record's address or identification is translated into the necessary physical records address(es). Since the logical record may be stored externally as one of many logical records within a physical record, or as one physical record, or as multiple consecutive physical records, or spanning multiple disjoint physical records, the address mapping function may be N:1, 1:1, 1:N, or N:M. The mapping function is defined by the implementation parameters for a particular logical file.

Since these mapping functions are implemented within the input-output module, all storage allocation, management, and storage level concurrent access control must also be implemented in the input-output module.

An input-output module supporting a logical interface may be implemented in one of two ways; either directly, using an input-output processor to off-load the bulk of the input-output module from the central processor, or indirectly, using one or more layers of software on top of a physical interface. For the purposes of the current discussion, the interface attributes, not the implementation details, are important. Consideration of an input-output processor implementation of a logical interface will be delayed until Chapter 5.

While all physical record interfaces are basically alike, the logical interfaces vary significantly with respect to the complexity of the relationships which are maintained automatically between logical records, the complexity of the available mappings between logical and physical record addresses, and the degree to which logical records may be reformatted during input-output.

Throughout this Section, the unqualified term 'record' refers to a logical record as defined in the logical storage structure.

3.5 Simple Logical Interfaces

Many logical interfaces simply provide minimal facilities beyond those associated with a physical interface (i.e. input-output without reference to a record's external physical address, and no more). These interfaces will be termed 'simple', and they feature:

(1) The mapping between physical and logical record addresses is 1:1 or 1:N.

(2) Either no record reformatting facilities, or simple (i.e. Fortran-like) format conversion for fixed format records. Consequently, even though the record's <u>location</u> is unknown, the calling procedure is not insulated from the <u>external</u> <u>format</u> and <u>structure</u> of the individual records.

(3) Access to individual records is either sequential, or via a <u>single</u> <u>record</u> <u>identifier</u>.

'Stream input-output' is one of the simplest sequential interfaces, whereby variable length unformatted records are supported using an 'end-of-line character' convention. The stream input-output interface forms the basis of the 'ports' mechanism (Balzer, 1971) which provides a uniform protocol for all process-process, process-file and process-device data transfers. Implemented examples include the ISPL

(Balzer, 1973) and UNIX (Ritchie and Thompson, 1974) systems.

Many of the abstract input-output objects discussed in Section 3.2 support simple logical interfaces to a calling program; e.g. the logical file abstraction. Interesti[illegible] Casey (1973) has proposed a very similar interface, ter[illegible] the 'logical data interface', however this proposal was motivated by the unused external processing potential of an IBM 3850 Mass Storage system, rather than the programming language and operating system structure concepts which led to the abstract input-output devices. The 'logical data interface' relies upon an external processor to manage the secondary storage space allocation and to implement the mapping from primary record identifiers into physical addresses -- all access requests to the input-output processor cite a logical file name and a value for the desired record's unique identifier.

The Multics system provides two different input-output interfaces, however they are both 'logical and simple'. The segmented-paged virtual memory architec[illegible]re is extended into the File System -- files are synonymous with segments -- to provide a uniform naming convention, hierarchic catalog structure, controlled access and sharing mechanism, and implicit input-output operations (i.e. demand paging) (Organick, 1972). Thus all the objects in the storage hierarchy which are visible to an executing process may be accessed in the same manner, whether they be procedure segments, program data segments or file data segments. From the programmer's perspective, a Multics segment forms part of a potentially very large address space which is constructed from fixed size logical records (i.e. the pages). Access to the non-storage devices (e.g. terminals, card readers and line printers) is supported by the Multics I/O System, via a set of stream input-output operations -- read, write and position. A segment within the File System may also be accessed as a 'pseudo stream device' through the I/O System (Feiertag and Organick,1971).

Some of the most common simple logical interfaces are those provided by the <u>file access methods</u> which have been either integrated or appended to most operating systems. These systems provide both file-level and record-level operations for files stored on secondary storage. Typical file-level operations include CREATE, DESTROY, OPEN, CLOSE, PERMIT ACCESS, etc., and involve maintenance of a global file catalog. This catalog is used to map a file name onto a set of physical areas of secondary storage, which need not be either continguous, or permanently allocated (i.e. a logical file spans one or more extents). Records within the files are stored and accessed according to one of a number of general file organizations; for example sequential, partitioned sequential, indexed sequential, fully indexed, or direct access. Table 3.3 lists the generic file organizations, along with some sample implementations, drawn from a number of common data management subsystems.

FILE ORGANZATION	SUBSYTEM ACRONYM OR NOMENCLATURE	HOST OPERATING SYSTEM
Sequential		
	-	Honeywell MOD 1 (MSR)
	BSAM	IBM OS/VS
	Sequential Files	MTS
Partitioned Sequential		
	BPAM	IBM OS/VS
Indexed Sequential		
	-	Honeywell MOD 1 (MSR)
	ISAM	IBM OS/VS
Fully Indexed		
	File System	Decsystem-10
	VSAM	IBM OS/VS
	Line Files	MTS
Direct Access		
	-	Honeywell MOD 1 (MSR)
	BDAM	IBM OS/VS

Sources: DEC (1975), Honeywell (1968), IBM (1973b) IBM (1973c), University of Michigan (1973).

Table 3.3 Examples of Some Common File Organizations and Data Management Subsystems

With the possible exception of the sequential organizations, certain field(s) of each record within a particular file are chosen to act as the 'primary key' fields, the values of which serve to uniquely identify each record.

The file access methods typically support three basic record-level operations, namely 'read', 'write' and 'position'. Read and write involve the transfer of one logical record, while positio is used in conjunction with sequential access to a subset of the records in the file. Associated with each operation is one <u>record identifier</u>, which identifies a single logical record within the file. For the fully indexed and direct access organizations, this identifier must be a primary key value. If the file is organized as sequential or partitioned sequential, the transfer operations use an implicit record identifer (i.e. 'the next record in primary key sequence'), but the position operation requires either an explicit primary key value, or a storage off-set (a number of characters or logical records, relative to the start of the file). Indexed sequential files require a primary key value for the position operation, while the read and write operations may use either a the 'next record in primary key sequence' identifier, or an explicit primary key value.

The distinction between physical and logical interfaces is not as clear for those sequential file organizations which permit access to the logical records on the basis of a storage off-set. Operations based upon this type of record identifier are sensitive to the sequence and lengths of logical records within the file, however they remain insulated from the physical allocation of the file's extents within the volume. In contrast, the 'next record in primary key sequence' and explicit primary key value identifier mechanisms are independent of any change in the record's position within the file or the file's mapping onto the physical volume.

3.6 Complex Logical Interfaces

The evolution of input-output support routines from data management to database management systems has seen the emergence of logical interfaces which are significantly more complex than those illustrated in the previous Section.

In comparison to the simple logical interfaces, these systems provide input-output interfaces which feature:

(1) Logical-to-physical address translations which range from 1:1 upto the most general N:M types of mapping.

(2) Flexible selection, construction and formatting for the fields within a logical record, including the synthesis of new logical record types from the fields of existing records.

(3) Access to logical records via a primary key, plus a variety of secondary keys and logical positional identifiers.

A typical database management system provides user processes with a wide range of services. The following list is an extended version of the database management system design objectives proposed by Snuggs, Popek and Petersen (1974):

(1) Data Independence: Changes may be made to the physical representation and organization of the logical records without impacting the correctness of programs which

access those records. This stability must prevail during record reformatting, record relocation and installation of different access methods.

(2) Support for a Global Data Model: One abstract model is used to describe the user's perception of how the data is organized and structured (i.e. what criteria are used when grouping logical records into logical files or subfiles, the classes of relationships which ~~may~~ be defined between logical records of similar or dissimilar types).

(3) Data Integration: All the data files for a particular suite of programs or corporate application are integrated into one data structure to minimize the unnecessary data redundancy and maintenance costs.

(4) Data Consistency: If a logical record is physically relocated or updated, the database management system assumes full responsibility for automatically performing the necessary secondary changes associated with indices, multiple copies of a record, free space lists and pointers, inter-record relationships, etc.

(5) Concurrent Access: The necessary locking mechanisms must be available to prevent interference between concurrent processes wishing to access the same data elements. These locks may be activated implicitly by the database management system, or explicitly by the active processes, however the incremental and unpredictable locking patterns associated with database applications

invariably necessitates some deadlock detection and roll-back / recovery facilities.

(6) Data Integrity: The database management system is charged with the detection and prevention of erroneous data values entering the database and/or incorrect inter-record relationships being established. The integrity of the database must be maintained during user generated updates in a concurrent access environment and following a system malfunction, due to either a hardware or software failure.

(7) Security: Assuming the users and owners of the executing processes have been correctly identified and authenticated in advance, the database management system must prevent unauthorized access to the data resources under its control.

(8) Access via Secondary Keys: Flexible facilities are available for the definition and subsequent automatic maintenance of such inverted lists and indices as may be necessary to provide acceptable performance for access to logical records via secondary key fields.

(9) Compatibility: The database management system should provide support for the current file organizations and access methods, to allow the user to access files which were previously maintained and accessed with the file access routines via the database management system.

To achieve these objectives, it is necessary to employ a multi-level scheme for describing the data structure.

This approach has been adopted in both the CODASYL and ANSI/X3/SPARC proposals for a standardized database management system organization (ANSI/X3/SPARC Study Group on Data Base Management Systems, 1975; CODASYL 1971,1973; Manola, 1976). These descriptions, or schemata, are used to formally describe the data objects and their int[illegible] relationships as they are viewed at differ[illegible] lev[illegible] in the implementation hierarchy. Typically the u[illegible] [illegible]hema is concerned with the user's 'conceptual' view [illegible] information structure, while a second schema describes the storage organization and access methods in a device independent context, and a third schema details the device level storage details (Bachman, 1975; Date and Hopewell, 1971; Nijssen, 1972; Palmer, 1974). The schemata are heavily utilized by the database management system to enforce integrity and consistency constraints, however their most obvious usage is as an incremental definition of the logical-to-physical address mapping for each logical record type. Since this transformation is effectively 'table driven', the mapping may be altered by modifying one or more of the schema definitions.

3.6.1 Alternative Data Structure Models

At the user level the desirable attributes for the data structure model in terms of which the conceptual schema is constructed, has provided considerable grounds for discussion, published papers, impassioned pleas and dogmatic disputes between the proponents of the various models. The principal models are termed 'relational', 'hierarchic', and 'network'; their origins may respectively be traced to the work by Codd (1970) on a mathematically complete set of construction rules and operators for data 'relations', the IMS database management system which is based upon an ordered hierarchy of sequentially accessed record sets (in other words, a tree) (IBM, 1974), and Bachman's influence upon the CODASYL Committee and his earlier association with General Electric's Integrated Data Store (General Electric, 1970) from which evolved the concept of connecting circularly linked sets to form a network. A comprehensive survey and comparison of these three basic approaches may be found in a special issue of the ACM's Computing Surveys (Sibley, 1976).

While these three models constitute the popularly accepted alternatives, many different and hybrid suggestions have been made. Some of these include:

(1) The formal definitions proposed by Hsiao and Harary (1970) for describing a wide range of list structured files and their associated access mechanisms.

(2) A 'Data Independent Access Model' (DIAM) which employs a multilevel hierarchy of submodels to support the (conceptual) Entity Set model in which the basic unit of information is the 'entity' and entities with similar properties are grouped into named 'entity sets' (Senko, Altman, Astrahan and Fehder, 1973).

(3) Separation of the 'entity' and 'relationship' concepts into two identifiable set classes forms the basis of the 'Entity Relationship' model proposed by Chen (1976). This model is very powerful at the conceptual level, since it provides a common basis for describing the implementation independent aspects of many other models (e.g. relational hierarchic and network).

(4) In attempting to provide some generalized database performance evaluation tools, Reiter 975) has developed a model based upon hybrid 'tree' and 'list' structures which may be used to describe both the logical structure and its physical realization.

(5) Hutt (1974) proposed the abstract 'data environment' as a general structural model which could be used by all user and operating system procedures to provide a uniform view of the data objects accessible to a process, and the valid manners in which they may be accessed.

(6) The 'information object' consists of a definition for a both a conceptual data structure and the permitted operations on that structure. Minsky (1974) has proposed this model as an extension of the (extensible) abstract data types found in some programming languages.

Despite the initial disagreements and the development of many alternative models, ther is an apparently growing trend towards the acceptance of a good deal of similarity between these models (Bac man, 1975; Olle, 1974,1975;

: bley, 1974). As Stonebraker and Held (1975) have suggested, this is particularly true once the models are analyzed at some common level which is independent of the degree to which the alternative data manipulation languages may be procedurally oriented. Consequently, the material in the later Chapters of this thesis is based upon the assumption that there are only two generic classes of complex logical input-output interfaces, namely the tabular, relational interface and the graph structured, network interface.

3.6.2 Alternative Database Management System Implementations

A database management system may be implemented in one of three ways:

(1) As an additional software package, placed on top of an existing data management subsystem. This is the approach most current implementations have followed, however there are some serious disadvantages associated with excessive processing overheads and poor security enforcement.

(2) By integrating many of the database management functions into the operating system. Whilst this approach has been suggested by Rodriguez-Rosell and Eckhouse (1977) as a possible future direction for the development of integrated software to handle data management, the only apparent implementation is the Data Base Access Method

described by Moriera, Pinheiro and D'Elia (1974). It appears that this approach may be doomed to failure in the long term, because the host operating systems are known to be insecure and unreliable -- the integration of the non-trivial database management routines into the operating system would only make matters worse.

(3) The advances in input-output subsystem architectures and external support processors may be utilized to off-load the database management system, or a large part thereof, onto a database processor. It is this type of implementation which will be investigated in Chapter 5, since it has considerable potential for resolving the performance and security problems, and at the same time enforcing the homogeneous input-output interface which will be introduced in Chapter 4.

Some of the input-output processors described in Chapter 2 have been constructed to provide direct support for logical operations which are compatible with one or more of the structured data models,[6] for example:

(1) RARES (Lin, Smith and Smith, 1976) and RAP (Ozakarahan, Schuster, and Smith, 1975) both support operations tailored for the non-procedural query languages associated with the relational data model.

6: Here, 'direct support' implies that, for the most part, the operation is executed outside the central processor, and an interface is available making these operations accessible to a normal user program.

(2) The initial backend database processors, XDMS (Canaday, Harrison, Ivie and Ryder, 1974) and DDM (Heacox, Cosloy and Cohen, 1975), were designed to process commands expressed in a CODASYL-like data manipulation language, for use with a network or data structured set model.

(3) Set Theoretic Information Systems (Hardgrave, 1975; STIS, 1976) have designed and are planning to implement special input-output subsystems hardware to support the primitive operations of the 'extended set theory' data model (Childs, 1968).

(4) Operations based upon flexible, query oriented data models for records with secondary key fields are provided by some of the database search engines, e.g. CAFS (Mitchell, 1976), CASSM (Copeland, Lipovski and Su, 1973), and the 'database computer' (Baum, Hsiao and Kannan, 1976).

(5) The proposed backend implementation of MRI's System 2000 database management system (Rosenthal, 1977b) would provide direct execution of data manipulation operations compatible with System 2000's underlying hierarchic data model.

3.7 The Effect of a Module's 'Level' within the System

The class of input-output operations directed towards the secondary storage devices originate from two principal sources. Either a user program explicitly requests a transfer (e.g. calls the input-output control system, or the database management system) or a resource management subsystem within the operating system initiates the transfer (e.g. paging operations in a virtual memory system, or transfers associated with a spooling device).

Performance considerations have historically led to the adoption of physical and/or, to a significantly lesser extent, simple logical interfaces for the operating system modules charged with executing input-output operations. However, there appears to be considerable variation between submodules in the some operating system. For example, within the Michigan Terminal System (MTS), disk input-output is performed by many modules, including the paging subsystem (PDP), the File System, the spooling subsystem (HASP), the on-line File Editor, the Program Loader and the compilers. Of these, the PDP executes channel programs from an absolute address space via an SVC to the operating system nucleus (UMMPS), the File System and HASP use different UMMPS supported SVCs to execute channel programs from a virtual address space, and the File Editor, Program Loader and most compilers use the logical file interface routines supported by the File System (McDonell and Marsland, 1977).

For user programs, things become even more varied. typically the many input-output support routines may be organized into a hierarchy, not unlike Gagliardi's extensions to THE, based upon the functional complexity of the input-output operations supported at each level -- the routines supporting the physical interfaces would be placed at the bottom, followed by the simple and then the complex logical interfaces. The wide variety comes not with the ordering within the hierarchy, but with the extent to which user programs interfaced to one level in the hierarchy have access to the interfaces at other levels in the hierarchy.

For a true 'virtual machine', or 'layered abstractions' hierarchy, a process is only aware of the facilities provided by the level immediately below the one at which it is executing, and not the details of the facilities and implementation of the lower levels. This is not the case in the input-output support hierarchy; for example, an OS/VS user program accessing a logical file via the IMS database management system may gain access to the same data via the VSAM file access routines, or directly via the 'execute channel program' (EXCP) facility.

3.8 Input-Output Facilities within Programming Languages

In this Section, the influence of various programming languages upon the available input-output interfaces will be discussed. The selected programming languages have been variously described as 'systems', or 'implementation', or 'applications' programming languages. Many of these languages provide access to the data management and file access routines described in Section 3.4.1, via standard subroutine or macro call. Some additional facilities may be provided by the primitive language constructs and the run-time input-output environment assumed by the compilers.

There are no intrinsic input-output primitives in the "C" programming language (Ritchie, 1973). However, the compiler supports a variety of calls to the routines of the underlying operating system, which in the case of UNIX (Ritchie and Thompson, 1974) support stream input-output and a positional operation (i.e. a call to 'seek').

The Espol language is an extension of Burroughs Algol, designed for writing operating systems and compilers (Burroughs, 1970a, 1970b). The Burroughs Algol run-time library supports a full repertoire of formatted, sequential read and write routines. In addition, Espol provides an intrinsic function (INITIATEIO) which gives immediate access to the basic hardware input-output operators (SCAN IN and SCAN OUT).

IMP72 (Bilofsy and Irons, 1973) and BLISS (Wulf, Russell, Habermann, Geschke, Apperson, Wile and Brender, 1971) have been designed for, and implemented on, a specific central processor and skeletal operating system -- the DECSystem 10. These two systems programming languages provide very similar, but limited input-output capabilities. In addition to calls across the simple logical interface supported by the underlying File System, the programmer may invoke the UUO Handler directly, to execute low level input-output operations. These operations are at the level of physical input-output, even though the channel and device names are symbolic.

The dialect of PL/1 in which a large part of the Multics operating system is written (Honeywell, 1976) features two basic input-output interfaces. The operators GET and PUT are provided for 'stream data sets', while 'record data sets' may be accessed by a group of standard read, write and position primitives. Record data sets may be accessed in either a sequential mode, or via a unique primary key value.

Very few attempts have been made to support the complex logical interfaces directly within existing systems programming languages (e.g. no common systems programming language provides a database management level interface to secondary storage). Within the INGRES database management system (Held, Stonebraker and Wong, 1975), a language preprocessor is used to allow the relational data sublanguage QUEL to be embedded within the programming language "C". However this approach has not been without its problems, principally related to the incompatibilities between QUEL and "C" and the need to defer essential consistency and validity checks until run-time, rather than performing them at compile-time (Allman, Stonebraker and Held, 1976). Wasserman (1976) claims that an integrated approach is required to develop a programming language and a data management interface in parallel, rather than grafting the necessary data management facilities onto an existing programming language. Another promising approach is the development of data definition facilities and data manipulation primitives which are independent of any particular conceptual data model, or host programming language. The Link Selector Language (Tsichritzis, 1976) and the proposals by Date (1976) for a general purpose 'database language' are tentative steps in this direction.

A significantly different approach involves supporting logical operations in an external data structure simply as an extension of the 'Multics-like' one level storage architecture into the programming language. This technique has been adopted in the non-Multics implementation of the MUMPS programming language (Bowie and Barnett, 1976) which treats all information objects uniformly, whether they are physically part of the program's local variables, or data elements within the external, tree structured file system.

The necessary input-output operations are executed by the MUMPS run-time system, and the programmer is unconcerned with individual secondary storage operations. A similar philosophy is evident in the use of an enhanced, PL/1 based 'structure' capability, to provide database structures which may be manipulated directly using the programming language facilities (Summers, Coleman and Fernandez, 1974). This general approach has been endorsed by Olle's (1974) contention that one likely trend for the "programmer's view' is a one level virtual storage, with the database management system handling the storage hierarchy management.

3.9 The Influence of Computer Networks and Distributed Databases

Increased acceptance of computer networks has meant that a program executing at one site may validly require access to information which is held on secondary storage at a remote center.

However, the access path to a given piece of information cannot be rigidly defined in advance. The actual sequence of operations will be determined at execution time, depending upon where the information is stored and in the event that the required information is not held locally, the status of the network. For user programs running under these conditions, low level input-output operations (e.g. channel programs and direct device control) are clearly impractical, and high level protocols must be enforced (e.g. the File Transfer and File Access Protocols for the Arpanet (Day, 1973; Bhushan, 1972)).

Consequently, network based programs requiring access to data held on secondary storage must phrase their requests in terms of 'network wide' logical record qualifications, rather than physical record addresses. All necessary access path resolution and transfer initiation will occur <u>outside</u> the calling program.

Coupled with high level information transfers between general purpose nodes in a network, there has been some development towards special purpose nodes for secondary storage only. The Datacomputer (Marill and Stern, 1975) is the prime example, providing other nodes in the network with facilities for the remote storage and efficient management of large volumes of information. Associated with the Datacomputer is a Datalanguage (Winter, Hill and Greiff, 1973), providing an environment in which a complex input-output interface could be supported across a network link connecting a host processor and a data management services processor. A similar approach is evident in the distributed database management system which is under development at Kansas State University, although the nodes in this network may act as dual purpose hosts for some processes and back-end database processors for other processes (Maryanski Fisher and Wallentine, 1976).

CHAPTER 4

A PROPOSAL FOR A HOMOGENEOUS INPUT-OUTPUT INTERFACE

Within this Chapter the structure of the input-output support software and the associated multiple input-output interfaces, described in the preceding Chapter, are critically reviewed. The factors which shall be considered include functional duplication, global software structure, resource utilization and reliability. From the shortcomings and disadvantages of existing approaches, a proposal is developed for the provision and enforcement of a single interface for all secondary storage input-output operations. This interface would be implemented using a single input-output module, whose internal structure and operation is not visible to routines using the facilities of the interface.

A number of alternatives exist for the choice of a single interface, however the one which will be advocated is derived from the CODASYL data description and manipulation proposals, which in turn assume an underlying network data model. This choice provides a logical, high level, programmer oriented interface, with many advantages for both user and operating system procedures.

The feasibility of the proposed homogeneous interface will be demonstrated using several examples based upon common input-output oriented modules within the operating system. Some conclusions will be drawn regarding the proposed interface's applicability for non-secondary storage devices, and a plausible scheme for 'phasing in' a software implementation of the homogeneous input-output interface will be presented.

4.1 The Case Against Heterogeneous Interfaces

For most software routines initiating input-output operations there is a high degree of conceptual equivalence between the transfer operations themselves. This fact forms the corner-stone of the justification for a single software interface to the input-output module, namely the observation that at least conceptually, multiple interfaces are not required. Further substantive evidence comes from the inefficiencies and cost of the alternative heterogeneous approach, e.g. unmanageable software structures, unnecessary duplication of support functions across operating system modules and inflexible device allocation strategies which are needed as a consequence.

4.1.1 Functional Equivalence

Functional equivalence between input-output interfaces may be demonstrated by showing that, despite the different communication protocols, underlying data models, complexity of physical-to-logical structural and address mappings, and other divergent factors, there is a degree on commonality with respect to the human conceptualization of "what an input-output operation involves". For example, most programmers would agree that, conceptually, there is no substantial difference between fetching a virtual page from backing store, retrieving a database record, or requesting the next record from a data set associated with a spooled device.

At this point, it is appropriate to identify the common attributes of the various interfaces which may be synthesized into a set of generic attributes for a functionally equivalent abstract interface. The following list is presented in rather a broad framework,[1] however, further refinements and additions will be made later in this Chapter:

(1) There is a unit of information which may be transferred between an immediately addressable main store buffer

1: These attributes are described in terms of a logical storage structure, however the same attributes apply to a physical storage structure -- merely substitute 'block' or 'physical record' for '(logical) record', and 'volume' or 'extent' for 'file'.

area and a secondary storage area which can only be accessed via an input-output operation (i.e. the concept of a record which is extern lly stored and internally processed).

(2) Records which have a similar structure, or are used for a similar function or application are grouped together and may be collectively referenced by a unique external name (i.e. the file concept).

(3) The unique identifier property ensures that individual records within a file may be unambiguously identified and accessed.

(4) If concurrent usage is possible, then some synchronizing and access control facilities are available.

(5) Rudimentary security facilities provide restricted access modes for particular users or user classes on a file-by-file basis.

It is a trivial exercise to demonstrate that all the input-output interfaces described in Chapter 3 possess these five attributes. An equally trivial exercise involves identifying the attributes which are *not* common. Most of the characteristics in the second group are related to the specific implementation details (e.g. 'how' a file is structured, 'how' a record is uniquely identified, 'which' access paths are available, 'what' types of access mode are available, etc.). However, there is sufficient evidence to establish the premise that *there is a good deal of commonality* between the various interfaces.

Once this functional equivalence across the input-output interfaces is acknowledged, a motivation is established for reviewing the current implementation strategies from the perspective of "what advantages, if any, accrue from the heterogeneous implementation of operations which are fundamentally similar?".

4.1.2 Software Structure and Duplication within the Input-Output Subsystems

It must be stressed that the near universal acceptance of heterogeneous input-output interfaces in current computer systems does _not_ constitute an _a priori_ justification for the approach as either optimal or desirable. As the discussions in Section 3.1.2 showed, the existence of multiple software interfaces is a historical anomaly resulting from the development of functionally related, but physically disjoint, support systems on top of low level physical input-output capabilities.

The input-output support subsystems described in Chapter 3 form a set of concurrent processes which interact to varying, and often obsure, degrees and provide a user process with a set of interfaces which feature a broad spectrum of communications protocols and functional complexities. Within a distributed processing environment, Jensen (1975) has expressed concern at the proliferation of heterogeneous interfaces between processes -- this trend invariably leads to unreliable systems which are very difficult to maintain and modify -- and there is no reason to doubt that the same general conclusions would hold for the concurrent processes charged with supporting the input-output interfaces, and the user or operating system procedures which use the interfaces.

By way of further evidence, "dynamic behaviour and communication between processes", and changes involving input-output related functions have been identified as two of the significant causes of programming errors during one series of modifications to the highly unstructured IBM DOS/VS operating system (Endres, 1975). The development and maintenance of reliable software for those modules and subsystems requiring input-output services would be simplified if the routines could interact with a single autonomous process (i.e. the input-output module) which is responsible for executing all secondary storage operations.

Within the input-output module itself, considerable advantages flow from the adoption of a single interface, specifically:

(1) The input-output module would be physically smaller (i.e. fewer instructions) than the current conglomeration of input-output support routines and their associated duplication of service functions, such as address [illegible], directory maintenance, channel program construction, concurrent access control, load

balancing, and storage management.

(2) Since the input-output module is only required to support one, invariant interface, internal reorganization could be more easily performed to provide a 'clean' structure for the component routines. This internal restructuring could proceed without impacting the routines outside the input-output module. Any of the commonly accepted programming methodologies (e.g. 'top down', 'structured programming', or whatever the locally popular choice might be!) could be used as a basis for the module's internal organization.

(3) As a result of (1) and (2) the input-output module would be easier to implement and modify, and the expected software reliability would be significantly superior to the present ad hoc implementations.

4.1.3 Resource Utilization

Many of the current input-output support subsystems are implemented using a low level physical interface, and minimal sharing of routines, even for the common functions. Consequently, these subsystems are unable to co-ordinate their concurrent storage management and access requirements. Typically the systems implementor is faced with no alternative other than statically dedicating entire devices to particular support subsystems.

The Michigan Terminal System presents something of a classic 'anti-example' in this regard -- the available secondary storage devices must be partitioned into three disjoint groups, one for each of the paging, spooling and on-line file subsystems.

There are many fundamental weaknesses evident in this approach to resource allocation:

(1) 'Load balancing' may only be applied locally, within a particular subsystem.

(2) For a heterogeneous collection of devices (e.g. a drum and disk hierarchy), the resource utilization is rather insensitive and non-adaptive to changes in the demands for resources between subsystems. For example, a frequently accessed file cannot migrate to the high performance drum if that device is under the control of the paging subsystem, irrespective of the degree to which the drum is under utilized.

(3) The unit of storage allocation (i.e. an entire device) is typically far too macroscopic. A subsystem may <u>have</u> to be allocated significantly more storage than actually required. For example, the allocation may provide an integral number of devices whose combined capacity is greater than or equal the largest conceivable demands by the subsystem for storage -- however, this may be several orders of magnitude large than the actual maximum, or the mean requirement.

(4) Software tends to be designed and written to exploit this static device allocation. As a consequence, the

routines are heavily device dependent and not tailored for easy reconfiguration following a change in the device assignment. Such a reconfiguration may be necessary for performance improvements, or during a hardware upgrade, or as part of a post-failure recovery procedure.

(5) As a whole the system is more vulnerable to device failure since there is little device redundancy, and no dynamic device swapping between subsystems.

For a a system featuring a single input-output module, <u>all</u> secondary storage devices would come under centralized control -- resource allocation, scheduling, and utilization could be optimized over <u>all</u> <u>subsystems</u> and over <u>all</u> heterogeneous <u>devices</u>.

To achieve full benefit from a single input-output module's capacity for centralized, dynamic resource allocation, it is necessary that the calling programs be insulated from any adverse effects associated with a change in the input-output subsystem configuration. For a logical interface, this is guaranteed since no program is aware of the physical addresses at which records are located. For a physical interface, a simple solution would involve translating the address parameters within the input-output module -- this mapping need not be very complex and is analogous to the 'virtual physical devices', supported by an IBM 3850 mass storage device (IBM, 1975) -- so the user's program 'believes' it is still dealing with physical devices and physical addresses.

4.1.4 Security and Reliability

The uniform interface supported by a centralized input-output module provides considerable potential for improving the system's global security and reliability. Security is enhanced directly, as a consequence of removing the multiple access paths users are currently able to exploit when attempting to retrieve or modify objects in the secondary storage environment. These same factors have been identified as one of the major advantages of current database management systems (Manola, 1975).

Typically, the security checks performed in the component support routines of a current operating system are not consistent, and as a result the one user may be simultaneously presented with multiple, different 'views' of the accessible secondary storage areas, depending only upon the particular input-output support routines which are used. When the same incomplete checks are performed in an environment in which potentially conflicting, concurrent accesses are being attempted, the situation becomes even less secure. The privacy and integrity of large sections of the external storage may be compromised by the relatively simple expedient of simultaneously updating one physical record using two disjoint input-output support subsystems.

If all input-output requests are passsed to a centralized module via a single interface, then the security enforcement procedures may be centralized and applied uniformly to all secondary storage accesses. This centralization of the <u>enforcement</u> mechanisms is highly advantageous, irrespective of the particular security <u>policy</u> which is to be implemented.

There appears to be considerable parallelism between this centralized security organization for controlling input-output operations, and the 'security kernel' organizations which are currently being investigated, with a view to constructing a basic security nucleus from which a secure operating system may be constructed (Popek and Kline, 1974; Schiller, 1975; Schroeder, 1975; Wulf, Cohen, Corwin, Jones, Levin, Pierson and Pollack, 1974). At least two systems have been proposed in which a secure data management system (i.e. an input-output module) would be implemented on top of an operating system security kernel -- the secure 'Data Management System' is based upon the kernel version of the Multics operating system (Hinke and Schaeffer, 1975), and a secure 'File Management System' has been implemented on top of a PDP 11/45 securitv kernel (White, 1975).

However, there appears to be sufficient variation in the design objectives and supported facilities to warrant divorcing the secondary storage protection mechanism from the operating system protection mechanism (Downs and Popek, 1977). The principle justification for this approach lies with the fact that the 'unique name property', upon which operating system security kernels rely, does not hold for secondary storage objects at the level at which users, or user processes formulate access requests. As an example, the operating systems objects such as processes, messages, address spaces, devices and processors all have unique names, and no two names fer to the same physical object. However even if unique names are assigned to a device, a volume, a file, a page and a logical record (via its identifier), the <u>logical</u> and <u>physical</u> objects which they refer to may not be disjoint. Thus the security policies for secondary storage objects may only be enforced if a single convention and mapping function is implemented for the conversion of non-unique and overlapping names by which users refer to data items and input-output objects, into a unique, system wide, nomenclature. The natural place for implementing this mapping is in the input-output module, since it requires access to the da[illegible]a definition schemata, and the mapping is also required w[illegible]in the module for the physical input-output operations, during logical record synthesis, and as a necessary prerequisite for providing a coherent scheme to control concurrent accesses.

One potential weakness of a single input-output interface and its accompanying input-output module is that the reliability of the entire system is dependent upon the reliability of the input-output module -- a variation on the time-honored 'weakest link' theme!. At this stage we are concerned with software reliability-- the reliability of the underlying hardware will be considered in Section 5.3. Given the importance of the heavily utilized input-output routines, several techniques are available to improve the module's expected reliability:

(1) If the module is to be substantially developed from scratch, there is considerable potential for starting with a software structure which is 'clean' and easily comprehended.

(2) Fewer errors per supported function would be expected, because the total code size will be smaller, and the interfacing between software routines should be standard (this applies both internally, for the component routines of the input-output module and for the input-output module's interface to the rest of the operating system and user worlds).

(3) The techniques of software redundancy which have been proposed (Randall, 1975; Melliar-Smith and Randall, 1977) are singularly well suited to the input-output support environment, where the concepts of 'acceptance test' and 're-try' are already evident. Note that although the input-output module presents a single interface to all calling routin , it may have considerable internal redundancy; e.g. performance considerations may dictate that alternative access methods be concurrently supported.

In any event, given that the input-output module will contain software errors, and that it will have to be modified and updated from time to time, and that fatal system failures may require extraordinary restart procedures, then there is an obvious requirement for some path to the input-output subsystem which bypasses some, or all, of the input-output module. Use of this special facility should be restricted, at least to one select group of systems staff, database administrators and operators. If the input-output module runs on a separate processor (refer to the next Chapter), it will be argued that use of this 'ultra privileged' access mode be further confined to a human interaction via the operator's console on the input-output processor.

4.2 Alternative Interfaces and Data Structure Models

The choice of an input-output interface influences both the structure of the central processor based software, and the range of feasible input-output subsystem architectures. However, the economics of current computer system implementations (Boehm, 1976) dictate that during the choice of input-output interface, hardware considerations should be largely subjugated in favor of software considerations -- any shift from the 'hardware status quo' should reduce the software development and maintenance costs. Consequently, the following discussion aims to present a software-oriented justification for the input-output interface design decisions. Once the interface has been chosen, attention will be directed towards efficient implementations of the input-output module (refer to Chapter 5).

In selecting a single interface for all secondary storage input-output it is necessary to choose an interface which is both 'natural' (i.e. not unduly artificial with respect to the operations which must be performed) and economically viable for the complete spectrum of applications supported by the current heterogeneous software interfaces. It must be stressed that adoption of a uniform interface has an effect which impacts routines at all 'levels' of the system -- for example, at the upper most level a subsystem implementing a non-procedural database query facility would use the same interface employed by the lowest level paging routines. In the following Sections, the physical and logical record interfaces are studied with respect to their ability to support a flexible and economic interface for all secondary storage oriented input-output.

4.2.1 A Physical Interface?

Obviously all operations could be carried out at the channel program level -- after all, this is the way most current systems are implemented -- however the resultant code duplication, device dependent procedures and program complexity renders this option increasingly unattractive from a software development and maintenance perspective.

In Section 3.3, the disadvantages of the channel program interface were presented from a software perspective. Given the scenario for the input-output subsystem architecture outlined in Chapter 1, the channel program interface is even less attractive, since it is not well suited to communication between a process and an external input-output processor or a database search engine (the overheads are simply too great to realize the full potential of these units, designed for highly parallel and autonomous operation).

By this stage, it should be obvious that a low level physical interface is not suitable as a homogeneous interface -- mere (!?) common sense dictates that channel programming should become <u>less</u>, and not <u>more</u> widespread!

4.2.2 A Simple Logical Interface?

Another alternative for a uniform interface is to adopt one of the simple logical interfaces described in Section 3.5, i.e. either 'stream input-output', or a logical file abstraction, or a file access method.

The most obvious advantage a uniform, simple logical interface would provide over a physical interface, is secondary storage device independence for all software outside the input-output module. (This is based upon the perfectly reasonable assumption that a storage off-set is not permitted as a valid record identifier.) A number of desirable system attributes would follow the achievement of such a device independence, namely:

(1) All secondary storage space allocation and management could be centralized in the input-output module, outside the 'knowledge' of the routines requesting input-output services.

(2) Processes using the interface would be insulated from any changes to the input-output subsystem hardware, or data migration between heterogeneous devices.

(3) The input-output related routines outside the module would be smaller, since many of the common lower level functions would be moved to the input-output module.

(4) Enforcement of security and controlled access procedures may be executed more efficiently if the objects requiring protection are logical files or logical partitions (i.e. 'areas') within a file --- it is significantly easier to validate a file access request when it is issued, rather than laboriously validating every channel program which is generated by the request. This saving is possible because the input-output module is the only place in which channel programs may be constructed.

Although these are significant improvements, the simple logical interfaces have some inherent disadvantages, one of the most important of which involves inadequate concurrent access facilities. Concurrent access is typically precluded, ignored (i.e. 'user beware'), or heavily restricted interactions involving these interfaces. Implementing an integrated database management system on such an interface would entail, either, adding further access control mechanisms outside the input-output module, or attaching and releasing the file or area once per operation. The first solution invalidates the design concept of a centralized access control mechanism, and the second imposes a prohibitive processing overhead. The problems associated with concurrent access become even more troublesome when an attempt is made to support 'transaction based' locking (i.e. spanning multiple requests to the input-output module), incremental lock requests, deadlock detection and roll-back. It is apparent that the primitive mechanisms provided by most simple logical interfaces to support concurrent access are not adequate for the more complex types of access control required in a database management system.

Despite their device independence, the simple logical interfaces do not provide any <u>access path</u> independence. This has two unfortunate side effects; firstly, the input-output module is not able to independently alter the internal organization of a file or data set to reflect an alternative access method (e.g. to improve performance). Secondly, once a change is made to the access method for a particular file, significant software changes must be made to all routines which access the reorganized data.

As a final point, it is apparent that neither the stream interface, nor the file abstraction, nor any single access method is adequate for the full range of applications which must use the interface. For example, how should a programmer design a routine requiring random access to several files, using the 'get' and 'put' stream input-output primitives, conversely, there is no obvious transformation which would permit sequential access using the operations provided by a typical hash addressed random access file method. Because the access methods and file organizations are visible at the input-output module interface, there would be considerable pressure to provide multiple, simple logical interfaces ...... and we're back to where we started!!

4.2.3 A Complex Logical Interface?

The last alternative for a uniform input-output interface is to adopt one of the complex logical interfaces. All these interfaces provide the same advantages mentioned in the previous section, in relation to their simpler counterparts. In addition, some of the disadvantages mentioned at that time are avoided when a complex interface is used.

Specifically, all the complex logical interfaces are designed for use in a highly concurrent environment, where uncontrolled access would cause serious security, integrity and reliability problems. Therefore, the concurrent access controls tend to be more sophisticated than those associated with the functionally simpler interfaces.

Access path independence is provided to vary extents for the different complex logical interfaces. Proponents of the relational data model stress this aspect of the relational interface, while systems supporting network data models generally provide less access path independence. By employing multilevel data description schemata, and a judicious choice of data manipulation and accessing primitives, there is considerable potential for insulating the calling routines from the details of the access methods and their implementation within an input-output module supporting either data model.

The range of operations provided by a complex interface is typically 'complete', in the sense that a programmer may define data organizations of a complexity which is suitable for a particular application, and be able to manipulate items within that data organization using the facilities of the input-output interface. In other words, the interface supports a 'superset' of the facilities required for the rational implementation of a whole range of applications.

If the premise is accepted that, in general terms, a complex logical interface provides a suitable basis for a homogeneous interface to secondary storage, then the next decision centers around which complex interface should be chosen. As discussed in Section 3.6.1, the choice is between the interfaces associated with the relational and network data models. Some of the relevant factors which should be considered at this point are:

(1) Procedural versus non-procedural data manipulation facilities.

(2) Flexibility in matching the requirements of a particular application to the available data structures and operations.

(3) Software stability in the face of data reorganization.

(4) Ease of embedding the interface into the necessary programming language(s).

(5) Dynamic versus static data definition facilities.

There are certain advantages associated with the adoption of an interface which has a procedural approach to record manipulation. The alternative interfaces designed for non-procedural access to tabular data structures (e.g. the query interface for a relational data model) are highly desirable for a non-programmer's conceptualization of the data manipulation functions. But, they are less suitable for an interface to central processor resident software, since this interface is basically procedural and must be comprehended principally by programmers.

As the examples in Section 4.4 will show, the network model is sufficiently general to support the range of data structures which are required for rational solutions to many of the non-database applications requiring secondary storage resources. For some of these applications, the transformations required to convert the appropriate network schema into a relational form are sufficiently involved, that the resulting data structure is no longer well suited to the programmer's requirements or conceptualization of the problem at hand.

There are two relevant aspects to data reorganization, namely changes in record format, and changes involving the access path(s) to individual records. Either of the complex logical interfaces insulates programs from the first class of reorganization. The relational interface prevents calling routines from relying upon the access paths by making them completely transparent. However, manipulations and accesses within a network environment typically require some access path information (defined in the program's subschema), but the extent to which a routine is dependent upon a particular access path varies from one program to another.

For the most part the applications programming languages which are likely to find common acceptance are procedural in nature (e.g. derivatives of Algol, or PL/1). As a result, a procedural input-output interface could be embedded more easily in the host programming language than a non-procedural interface. Again, this situation favors the network data model.

For most applications, it is extremely unlikely that new data definitions will be constructed during a program's execution. (Although changes in data definition may be introduced between consecutive executions of the program.) For those applications in which non-programmers interact via a relational query language, the ability to create new relations (i.e. define new data structures) is essential (Boyce and Chamberlin, 1973; Chamberlin, Astrahan, Eswaran, Griffiths, Lorie, Mehl, Reisner and Wade, 1976; Date and Hopewell, 1971), however the requirement for this facility is less obvious for a software interface.

From the foregoing discussion, it appears that a procedural interface based upon a network data model, provides the best choice for possible adoption as a homogeneous software interface to the secondary storage resources.

4.3 The Uniform Input-Output Interface

There are two components of a complex interface, which between them define the functional capabilities of the interface -- one component is concerned with the definition of the elementary data items, the criteria for aggregation into sets, the relationships between sets, and the logical relationship between the members of a set. The other component specifies 'how' the data items and sets may be accessed and manipulated.

The CODASYL Data Base Task Group and its more recent off-spring -- the Data Description Language Committee and the Data Manipulation Task Group (within the Programming Language Committee) -- have spent the past ten years attempting to produce an interface to a network data model which would find common acceptance and broad manufacturers' support. Thus far, much progress has been made, but consensus seems a long way off. It is in this context that the decision was made to avoid proposing a new interface as part of the current research. Instead, I have opted to use the CODASYL proposals as a base from which modifications could be proposed, in order that the particular requirements of a uniform input-output interface may be achieved. While this decision probably does not constitute "standing upon the shoulders of those that have gone before", I feel it is at least an attempt to "get off their toes"!

It should be noted that while changes to the CODASYL proposals will be suggested in the following Sections, they are generally of a <u>minor</u> nature and tend to move the CODASYL interface towards the 'conceptual schema' level of the proposed ANSI/X3/SPARC database architecture. This has been achieved by omitting, from the input-output module interface, those CODASYL facilities which have been designated for the 'internal schema' level in the ANSI proposal. It is also possible to hypothesize that the various CODASYL committees are moving in a similar direction, based upon the pending changes and revisions to the various data definition and manipulation language specifications.

One initial disadvantage of working with the CODASYL proposals is the COBOL orientation and syntax of the Data Definition Language (DDL) and the only Data Manipulation Language (DML) fully specified to date. However, this will be overlooked, since the available facilities of both the DDL and DML could readily be translated into a semantically equivalent syntax which is more readily suited to a general purpose implementations programming language (e.g. DML verbs could be mapped onto Algol-like procedure calls, and DDL clauses are readily transformed into data structure declarations). Henceforth, the syntax will be ignored, and attention directed towards the semantics of the DML and DDL facilities.

Throughout the following discussion, it will be assumed that the reader is familiar with the CODASYL concepts, in particular, data item, database-identifier, database-data-name record type, record occurrence, set type, set occurrence, and run-unit currency.

4.3.1 Data Definition Facilities

The CODASYL Committees have published proposals for two data definition languages; the schema DDL (CODASYL, 1973) and the COBOL subschema DDL (CODASYL, 1971). The following comments will be based upon the schema DDL, given the assumption that this is a good approximation to the desired input-output module interface. It is anticipated that this interface will support the 'middle' layer of at least three levels of data description. One, or more, lower levels will appear within the input-output module to describe the mapping of the uniform interface data items onto the physical storage media. An optional higher level of data description would be implemented within those applications supporting non-procedural data manipulation and definition facilities for end-users (i.e. the user interface for non-programmers). Each user or process will only have access to those components of the interface schema for which expicit usage permission has been granted by the database administrator.

Most of the schema DDL facilities are adequate for the requirements of the uniform interface. With specific reference to the generic attributes described in Section 4.1.1, the unit of information transfer is a record, records of similar type or usage may be grouped together into sets, records must be uniquely identified, at least within each record type -- refer to the comments later in the Section for further details -- and security facilities are provided via the privacy locks[2] associated with the schema, set and record description entries.

4.3.1.1 Record Description

The DDL record description facilities require some minor changes to meet the uniform interface requirements. In particular, the location mode clause should be abandoned, and a mandatory unique identifier declaration introduced.

Since the location mode clause is used "to control the assignment by the DBMS of database keys to records", it should be removed from the uniform interface schema, and subjugated to the lower level storage description within the input-output module. This decision is based upon consideration of the improved access path independence which would be provided for the routines using the interface. A promising advance on the part of the Data Description Language Committee (DDLC) is the apparent removal of all references to the database key in the DDL (Manola, 1976). As with the location mode clause, this detail should not be visible to interface users. In a similar vein, the use of areas has been excluded from all the proposed interface facilities, since the operations and descriptions should

2: Manola (1976) reports that the term 'privacy lock' will, in all likelihood, be replaced by the more meaningful term 'access-control lock', when the updated DDL specifications are published.

deal entirely with the logical entities -- data items, records and sets. Consequently, the within clause of the record subentry would also been dropped.

Currently, a unique identifier may be defined implicitly by appending the 'duplicates are not allowed' phrase to either the location mode is calc clause of the record subentry, or to one or all of the member, key, order, and search clauses of the member subentry (Nijssen, 1975). It appears likely (Manola, 1976) that the pending action of the DDLC to add of an optional identifier clause within the record subentry will partially solve the unique identifier problem by making the declaration explicit -- the only additional change proposed here is to make the declaration mandatory, and enforce uniqueness either for all occurrences of the record type, or for all records of the same type within a particular set occurrence. A possible declaration may be of one of the forms:

(1) IDENTIFIER database-identifier-1 [, database-identifier-2]

(2) IDENTIFIER database-identifier-3 [, database-identifier-4] WITHIN set-name-1 SET

Note, more than one declaration of the second format may be specified, up to a maximum of one declaration for each set type in which the record type may be a member.

4.3.1.2 Set Description

With the possible exception of allowing records of the same type to be both the owner and members of a set, the restrictions upon multiple set membership and singular set ownership are deemed to be <u>not</u> <u>unduly</u> restrictive.

The proposed addition of a <u>fixed</u> set membership option to the existing <u>mandatory</u> and <u>optional</u> modes (Manola, 1976) is considered to be a positive step.

For similar reasons to those proposed in favor of dropping the <u>location</u> <u>mode</u> clause, the <u>order</u> and <u>key</u> clauses defining the sequence of member records within a set should be omitted from the interface schema. The sequencing of records within a set is considered to be a data <u>manipulation</u> function (refer to the ORDER DML operation), and not appropriate for the data description, if the interface is to support access path independence. As far as an interface user is concerned, records within a set may be accessed randomly using the identifier, in identifier sequence, or in a user determined sequence (provided the records have been explicitly sequenced with an ORDER DML operation).

Perhaps the most contentious aspect of the CODASYL DDL is the <u>set</u> <u>selection</u> mechanism, used to describe <u>which</u> set occurrence a member record should belong to. Olle (1975) and Manola (1975) are amongst those who have attempted to clarify this facility, however the issue remains very confusing. It appears that the original specifications contained too many concepts which were only peripherally related to set selection.

Recently, Nijssen (1975) has suggested replacing the whole selection clause and its five alternative formats, by a single clause which would define set membership solely on the basis of equality between the identifier of an owner record and nominated data item(s) in the member record. This approach would be functionally equivalent to the existing options, but it is conceptually much simpler. But mandatory imposition of this scheme would cause significant increases in data redundancy, unless the input-output module is 'smart enough' to remove the common data items from the member records prior to storage, and then re-insert the value prior to passing a retrieved record back to the user process.

The DDLC's response to these suggestions has been to add Nijssen's proposal as a structural constraint, to be included in the set definition, and then allow the set selection to be made upon the basis of a structural constraint. This option has been added to the existing set selection criteria.

A further refinement of the set selection clause would involve replacing both the database-key and calc-key options with a selection criteria based upon a supplied value for the desired owner record's identifier. The rationale here is to try and remove those DDL components which are dependent upon the input-output module's internal operation, or the actual placement of records in secondary storage.

For the homogeneous interface schema the following basic modes of <u>set</u> <u>selection</u> are proposed:

(1) THRU database-identifier-1 IN <u>OWNER</u> <u>EQUAL</u> TO database-data-item-1 IN <u>MEMBER</u>

(2) THRU <u>COMPUTED</u> database-identifer-2 FOR <u>OWNER</u>

(3) THRU <u>CURRENT</u> OF OWNER

(4) THRU <u>SYSTEM</u>

It appears that these four options provide sufficient scope for clearly defining set membership, according to one of several different criteria -- all of which are independent of the details of record storage and placement.

4.3.1.3 Consistency and Integrity

At some point in the data description process, provision must be made for the definition of assertions and constraints to control the validity, integrity and consistency of the secondary storage resident data. Currently, some facilities are dispersed through the <u>record</u> and <u>set</u> description entries, however a preferable approach would involve placing all these constraint definitions in one special section of the <u>schema</u>.

These definitions impose limits upon the valid ranges for data item values, permit restricted classes of relationships between data items or record occurrences, and generally ensure that the stored data presents as accurate a description of the corresponding real world entities as possible.

Whenever a change is made to the stored data which may affect one of the defined constraints, a checking procedure is invoked before the change is made to verify that no violation of the constraints will occur (e.g. the proposed 'trigger subsystem' (Eswaran, 1976)).

4.3.2 Data Manipulation Facilities

The 1971 Data Base Task Group report outlines the DML facilities suitable for a COBOL host language environment. For each of the primitive operations, the following list describes the general semantic intent of the operation (i.e. its meaning, independent of the COBOL framework), and any modifications deemed necessary for the implementation of a homogeneous input-output interface.

(1) OPEN (and CLOSE): specify an intent to use (or release) a portion of the database. The subject of this operation should be one or more <u>interface</u> <u>schema</u> <u>names</u>. By adopting a more powerful concurrent access control mechanism, it should be possible to omit the <u>usage</u>-<u>mode</u> clause from the OPEN operation. OPEN and CLOSE are used principally to perform internal house-keeping within the

input-output module (e.g. buffer allocation, user work area initialization, and establishing the necessary mapping functions or tables to achieve the structural and address transformations between the various levels of data description).

(2) KEEP (and FREE): an inadequate concurrent access control mechanism which notifies the requesting process after an interferring update has occurred. Section 4.3.2.1 outlines an alternative LOCK / UNLOCK mechanism designed to replace the KEEP / FREE operations.

(3) INSERT (and REMOVE): following an addition or an update of a record, modify the record's set membership for those sets in which membership is not defined to be automatic.

(4) STORE (and DELETE): add (delete) a record, and make all associated changes to sets in which the record is either the owner, or an automatic member (STORE), or any type of member (DELETE). When a new record is added, the input-output module must also create new empty set occurences for each set type of which the added record type is defined as the owner record. If the new record is the member of a set with a selection clause involving the thru computed identifier option, the STORE operation must have an appended 'USING database-identifier = <expression>' phrase, so that the record's correct set membership may be determined. Deletion of a record which is the owner of a non-empty set causes various

secondary deletions to occur amongst the member records -- all members which are mandatory members are deleted, optional members are removed from the set and optionally deleted from the database. The input-output module is responsible for automatically executing the necessary space allocation and / or reclamation, caused by the addition or deletion of a record occurrence.

(5) MODIFY: update an existing record, and perform any necessary set membership changes.

(6) FIND: search for a specified record, on the basis of the requested record's set membership or ownership, a record identifier, or the next / prior / first / last / nth record[3] in a particular set. Section 4.3.2.2 presents the outline of a more general FIND operation, capable of locating more than one record occurrence.

(7) GET: retrieve (i.e. transfer from secondary storage to the user work area) a record which has been previously located using a FIND operation.

(8) MOVE: saves the identifier of the current record for a particular record or set type in a local, temporary save area.

(9) ORDER: logically resequence the member records of a particular set. This operation would normally precede a sequence of GETs following a FIND which located more

3: The set sequence defaults to 'identifier sequence', however this may be changed by a previous ORDER operation.

than one record, or prior to a FIND operation in which the desired record was to be located positionally within the set (e.g. FIND NEXT, or FIND FIRST).

(10) IF: test the emptiness of a set, or the current record's ownership or membership of a set.

(11) USE: determine the status of the previous DML operation.

4.3.2.1 Concurrent Access and Locking

As many writers have pointed out, the CODASYL KEEP and FREE operations are inadequate for a concurrent access environment (Hawley, Knowles and Tozer, 1975; Robinson, 1975; Schlageter, 1975; Shemer and Collmeyer, 1972).

Despite the work by Papadimitriou, Bernstein and Rothnie (1977) into defining classes of transactions which may execute concurrently without any locking or mutual interference, the vast majority of applications involving shared secondary storage resources require some explicit locking mechanism.

The principal issue in discussions concerning LOCK / UNLOCK primitives seems to be the definition of 'what' should be locked, and 'how' it should be described. In general terms[4] the options appear to be:

4: The locks considered here relate to <u>logical</u> objects, hence the physical locking mechanisms (e.g. Univac's DMS 1100 (Robinson, 1975)) have been omitted.

(1) Lock individual record occurrences using the record identifier (Shemer and Collmeyer, 1972).

(2) Lock individual set occurrences by citing the set name and the owner record's identifier.

(3) Lock multiple records of the same type by specifying desired values or ranges for particular data items within the records; e.g. Schlageter's 'lock by value' (Schlageter, 1976), or the 'predicate locks' proposed for System R (Eswaran, Gray, Lorie and Traiger, 1976).

Of these alternatives, the last is by far the most elegant, however its implementation may be impractical given the current state-of-the-art -- IBM's System R project has since reverted to a less rigorous locking procedure based upon varying 'degrees of consistency' (i.e. the extent to which you are willing to permit other users interfer [illegible] your processing) and varying lock 'granularities' (i.e. locks may apply to sets, records or fields within a record) (Gray, Lorie, Putzolu and Traiger, 1976).

Hence the current proposal to support <u>both</u> of the other locking options. This constitutes a realistic solution which should prove adequate for the majority of applications.

A lock may be explicitly requested (released) using the DML operation LOCK (UNLOCK). Once granted, as far as the user is concerned, this lock will remain in effect until it is explicitly released. Internally, the input-output module will apply implicit locks, using the same mechanism, for the duration of single DML operations which modify the state of the secondary storage resident data.

Related to concurrent access and locking protocols are the twin problems of deadlock detection and deadlock recovery. It is assumed that the controlled procedures which are designed to avoid deadlocks (e.g. requesting all locks simultaneously, or imposing a conceptual ordering upon the available locks (Sekino, 1975)) are not suitable for a general purpose input-output interface. Hence the input-output module must provide the necessary procedures to support checkpoint, roll-back and restart facilities. Whilst this is a costly service to provide, it is unavoidable -- one compensation is that a substantial body of literature has evolved concerning 'how' it should be done (e.g. King and Collmeyer, 1973; Macri, 1976; Schlageter, 1975; Shemer and Collmeyer, 1972).

4.3.2.2 The FIND Operation

Current DML specifications indicate that the result of a FIND operation can be _at most_ one record. In view of the potential for parallel processing in the input-output subsystem, this type of one-record-at-a-time retrieval seems unduly restrictive. For example, it would be impossible to make efficient utilization of a RAP-like device using operations which search for a single record.

The alternative is to permit the FIND operation to generate a pseudo-set of zero, one or more records (call this set the FIND-SET for the moment). Subsequently, a GET operation would refer to the next record in the most recent FIND-SET (the record sequence within the FIND-SET could default to identifier sequence, however further flexibility could be added by permitting the ORDER operation to be performed on the FIND-SET). In this manner, the input-output module interface could provide a set retrieval capability which could be implemented, either in software, or using a database search engine.

It is proposed that the record selection expression of the FIND command be a Boolean combination of relational subexpressions. Each subexpression would contain a database-identifier (or database-data-item), a relational operator ('>', '<', '=', etc.) and a value or local data-name containing an appropriate value. This format could be used to replace the 'FIND record-name USING database-key-name' format of the current DML.

4.4 Input-Output Operations Involving Secondary Storage

Having established the desirability of a homogeneous interface between central processor resident programs and the secondary storage subsystem, and defined an appropriate interface, the feasibility of the approach remains to be demonstrated.

A necessary precondition for the imposition of the uniform interface is that all secondary storage input-output functions can be effectively implemented using the facilities of the proposed interface. The following sections show that in particular, this precondition can be achieved for those components of the operating system which currently do not perform secondary storage input-output via a complex logical interface.

4.4.1 The File Sysytem

For a typical time-sharing system (e.g. the Michigan Terminal System (Pirkola, 1975), or the UNIX system (Ritchie and Thompson, 1974), the general purpose file system could be readily implemented on t of the proposed input-output interface.

The structure of the necessary user directories and assorted file types can be simply mapped onto the conceptual information structures supported by a network data model. Thus, the user's view of the file system would not be impacted by a change within the support software which saw the present low level input-output operations, replaced by calls to the input-output module.

All code concerned with the physical devices could be removed from the file system routines (e.g. device dependent input-output operations, the allocation/management of secondary storage space and device buffers, record blocking and unblocking, and input-output request scheduling). The remaining file system functions -- 'file name' directory maintenance and searching, accounting procedures, and file level control over concurrent access to shared files -- would be implemented using the interface facilities. Following minimal intervention on the part of the file system, user requests for file system operations would be transferred to the input-output module for execution.

4.4.2 The Spooling Subsystem

In multiprogramming environments, spooling subsystems are frequently employed to maximize device utilization and to reduce the input-output wait time for programs accessing the slower, unit record devices.

In a typical implementation, user processes which require access to a spooled device interact with a pseudo device, which is in fact a file held on secondary storage. Outside the user process's view, the spooling subsystem is responsible for transferring information between the physical devices and the files corresponding to the pseudo devices.

Appendix A contains the preliminary designs and descriptions of a spooling subsystem implemented on top of the proposed interface. This example lends substantial justification to the uniform interface proposals, in as much as the viability of implementing a non-trivial subsystem, using the interface facilities, has been demonstrated. The following resume is derived from the material in that Appendix, and the experience gained during its development:

(1) Processes which have historically executed input-output operations at the lowest level can readily be transformed to perform input-output at a much higher level. Such a change does result in smaller software modules, the advantages of which were covered in Section 4.1.

(2) At the point where user processes intiate their input-output requests it is immaterial whether a data file is spooled or permanently resident on secondary storage, since both classes of files 'look alike' to an executing user process.

(3) None of the processes which interact during spooling operations need be aware of the physical representation or structure of the spooled data on secondary storage. Again the advantages are adequately covered in Section 4.1.

4.4.3 The Paging Subsystem

Like spooling, the paging function in most virtual memory systems has been implemented using special software for handling the necessary input-output transfers involving the dedicated paging device(s).

The potential barriers to adopting a high level approach to paging input-output appear to lie with efficiency considerations, namely, can paging operations, performed via a database management system, be executed with sufficient speed to ensure that the central processor is not excessively 'idle'? These issues will be discussed in Chapters 5 and 6 when a database processor implementation of the proposed input-output module will be described. At this stage, it appears that the efficiency question can be resolved, without compromising the conceptual foundations of the homogeneous input-output interface.

Another problem associated with a high level invocation of paging input-output is buffer definition. When a program requests service from a database management system the buffer location is typically outside the caller's direct control (e.g. programs written in COBOL). For paging operations, the paging subsystem must be able to define a buffer location which ranges over the entire real address space. A simple solution to this problem would involve permitting all programs to optionally specify alternate buffer locations within their virtual address space, and to equate the virtual address space of the paging subsystem to the real address space. As with conventional operating systems, the nucleus or kernel must assume responsibility for translating the virtual buffer address into a real buffer address and 'locking' the buffer in main store before initiating a physical transfer operation.

Provision of this type of alternative buffer facility would also prove useful for those applications which require many occurrences of a particular record type to be resident in main store, for example an on-line file editor.

4.5 Input-Output Operations Involving Sequential Devices

Clearly a complex logical interface is well suited to the most active and expensive class of input-output devices -- the conventional, random access secondary storage devices. However, the remaining devices are basically sequential in nature, and may be classified into three generic groups;

(1) unit record devices (e.g. line printers, card readers),
(2) terminal devices (e.g. teletypes, VDU's), and
(3) very large capacity devices (e.g. magnetic tape in current systems, or mass storage devices).

If the proposed homogeneous input-output interface is to be truly device independent, then the interface to the sequential devices should, to the maximum feasible extent, be the same as the interface to the non-sequential devices. This consistent appearance must include both the r [illegible] toire [illegible] avaliable operations, and the interface communication [illegible] ocol.

[illegible]t [illegible]e[illegible] that the proposed input-output module interface [illegible] e used for all spooled devices, archival devices and mass storage subsystems. On the other hand, the interface is apparently not well suited to the support of communications equipment (i.e. terminals), and, by analogy, the non-standard real-time devices (e.g. in control applications).

In a typical configuration, the unit record devices would all be spooled, and once the spooling subsystem is implemented using the homogeneous interface, then user processes may access the spooled files via the same input-output module interface (refer to Section 4.4.2 and Appendix A for further details).

4.5.1 Terminal Input-Output and the Communications Subsystem

Based upon the current state-of-the-art in communications and terminal-network systems engineering, it is highly likely that there will be some processor capability within the communications subsystem, but external to the central processor. Following the discussions in Chapter 2, it is clear that the actual processing may be done within a front end communications processor and/or using a small processor in the terminal itself (e.g. the HP 2640 communications terminals). This external processing capability means that, irrespective of the chosen software interface, much of the input-output processing related to terminal operations will occur outside the input-output module which supports the proposed uniform interface.

Terminal input-output operations are inherently different to secondary storage operations, as illustrated by the following list of attributes, generic to the terminal environment:

(1) Terminal users are typically involved in an <u>interactive</u> dialogue with the operating system, or an applications program.

(2) Most users try to limit their terminal output to as small a volume as possible.

(3) Differences between the response times and transfer speeds of terminal and secondary storage devices are typically in the range of 3 or 4 orders of magnitude.

(4) Messages input from, or output to, a terminal are not usually re-accessible once they have been sent, and are usually processed and discarded once they have been received. By comparison, a record written to secondary storage may be re-read as many times as necessary.

(5) There is a high degree of variability in message lengths and formats for terminal input-output, even within a single application.

(6) The terminal user is able to asynchronously interrupt data transfers, or gain the attention of the central operating system (e.g. the 'BREAK' facility).

(7) Transmission problems and syntax errors force the system to provide the user with an immediate correction, recovery or retry procedure.

When all these differences are taken into consideration, it appears that the uniform interface is not particularly suitable. Terminal support could be more readily provided using a 'stream input-output' interface which was physically and logically divorced from the proposed uniform interface. From a global perspective, the communications and secondary storage subsystems would operate independently, and with considerable internal autonomy, since each subsystem would support its own uniform interface to the resources under its control.

4.5.2 Magnetic Tape, Archival and Mass Storage Input-Output Operations

Very large capacity, serially accessed tertiary storage media have conventionally been used for three principal purposes:

(1) archive of inactive files,
(2) storage of active files which are too large for secondary storage, and
(3) as a medium for the transfer of data and programs between different computer installations.

All these applications may be efficiently supported using the uniform input-output interface, in conjunction with an 'intelligent' mass storage subsystem (refer to Section 2.6). Assuming that the mass storage subsystem is under the general control of the input-output module and, in particular, is not directly accessible from outside that module, then:

(1) Archival functions may be handled automatically by the input-output module in co-operation with the mass storage subsystem. In addition, inactive files may be automatically staged back to secondary storage when, and if, they are required. In other words, no explicit user action is required to archive, or restore files as their usage demands fluctuate.

(2) Since secondary storage is primarily dedicated to active files, a very large file could be made secondary storage resident for the duration of its use (i.e. inactive files which currently occupy large portions of the secondary storage resources may be archived to make more room available for the active files). Alternatively, very large files could be segmented and staged to secondary storage one segment at a time.

(3) A simple utility program could be provided to load / unload files between a magnetic tape device and the secondary storage areas under input-output module control. In this manner, the transfer of standard format tape reels between installations may be supported, but only one system utility process requires a non-standard magnetic tape unit interface.

Consequently, those routines requiring access to files currently held on tertiary storage may use the uniform input-output interface, and the input-output module will assume responsibility for ensuring that the requested file is staged to secondary storage, if it is not already there.

4.6 Implementation Logistics

Any proposal for a new software architecture is doomed to failure if it cannot be implemented within the budgetary and personnel constraints of those organizations capable of supporting large scale software production. Consequently, a few remarks will be made concerning feasible approaches for the introduction of a homogeneous input-output interface to secondary storage.

Firstly, no software in current usage is so error-free, adaptive and efficient that it will never have to be re-written, and hence redesigned. By themselves, the arguments raised in this research are probably not sufficient to <u>force</u> the development of new operating systems w radically different structures for the input-output support software. Rather, the objectives must be viewed from the perspective of defining the desirable attributes for replacement systems <u>when</u> the changes are made, for whatever reason.

In all likelihood, the strategy would have to involve a phased introduction, rather than a unilateral imposition of the new regime. Clearly the input-output module must be built first, either from scratch or based upon an existing database management system[5] -- the essential criteria at this early stage would be to introduce something which supports the interface.

Once the input-output module is operational, those programs and subsystems requiring secondary storage access may be modified one at a time, or as they are updated and replaced. As mentioned before, the internals of the input-output module may be revamped to reflect performance improvements, withdrawal of the underlying interfaces from direct usage, or dedication of more programmers to the project. The development of software to use the interface and changes to the input-output module may proceed in parallel, since the interface remains invariant.

As the heterogeneous interfaces become less popular and less heavily utilized, they may be withdrawn from public usage. If they are used by the input-output module (e.g. the channel program interface), then the support routines would be absorbed into the input-output module, otherwise the support routines may be abandoned.

For the input-output module to function correctly, it

5: The choice of a CODASYL based interface would assist this early implementation phase, since considerable knowledge and expertise is available concerning 'how' a CODASYL compatible database management system should be implemented -- this would less likely for a relational interface, and presents an almost insurmountable obstacle for the intrepid soul attempting to develop a 'home grown' interface.

must-be able to ensure that calling processes do not by-pass the uniform interface, or corrupt the tables and code within the input-output module. This implies protected entry points and controlled access to shared code and data, in addition to the conventional address space separation techniques. It is interesting to note that these very mechanisms could be supported with minimal changes to existing hardware and software; e.g. even the IBM Series/370, with its notoriously poor protection architecture, could support these facilities provided all programming was done in a high level language, like PL/1 (Fernandez, Summers, Lang and Coleman, 1976). An alternative technique for providing the necessary ancillary protection -- by implementing the input-output module in a separate processor -- will be discussed in Chapter 5.

The last consideration associated with the introduction of a uniform interface involves the available compilers and assemblers. Obviously, all the programming language facilities which were previously available and in contravention of the uniform interface philosophy would have to be removed, or replaced -- this is especially true of the physical interfaces included in assembly languages and many systems implementations languages.

CHAPTER 5

THE HOMOGENEOUS INTERFACE AND AN AUTONOMOUS SECONDARY STORAGE PROCESSOR

Moving the software components of the input-output module (described in Chapter 4) from the central processor into a separate input-output processor is the final step in the development of the global system architecture which has been the objective of this research. Since the input-output module will be involved principally in accesses to the secondary storage devices, the special processor shall be termed a <u>secondary storage processor</u>.

The justification for the proposed configuration will be largely qualitative in nature, based upon achievement of global design goals, a desirable software structure, and economic, performance, and security criteria. Of the points raised in favor of the overall approach, some relate exclusively to the imposition of a uniform input-output interface, some relate to the use of a dedicated secondary storage processor, and some relate to the cumulative effects of implementing a uniform interface on a dedicated processor. Arguments pertaining to the first category were discussed in Chapter 4, while the balance of the justifications will be presented in this Chapter.

5.1 Some Global Design Objectives

The overall structure of a general purpose computing system is influenced by a set of universally accepted global design objectives, which include:

(1) Meeting the requirements for increased privacy protectio[illegible]ted by new legal considerations and increasing [illegible] from both the users and the society at larg[illegible]

(2) Providing greater system reliability because,

(a) reliability is a necessary prerequisite for trustworthy enforcement of privacy policies,

(b) technological advances have made hardware architectures based upon 'fault-tolerant' and 'fail-soft' designs economically viable propositions, and

(c) the penalties associated with the operation of an unreliable system (i.e. real financial loss, user discontent, poor utilization of resources, etc.) will become increasingly unacceptable.

(3) Achieving a more rational approach to the structuring of complex pieces of software since,

(a) it is difficult to achieve the desired reliability with software structures based upon design philosophies which consider the code to be a single 'amorphous mass',

(b) the alternative approaches are less attractive with respect to software development and maintenance costs, and

(c) as firmware and hardware techniques make greater advances into traditional areas of software implementation, there is an urgent need to define and understand the role of specific software modules in relation to their operational environment. This requires an investigation of functional dependencies, interfaces, and the flow of both data and control -- a task which is greatly simplified if the pieces of software display some coherent structural basis.

(4) Maximizing the productive utilization of the physical resources, especially those which are most expensive to purchase and operate.

(5) Providing increased throughput and flexibility, along with reduced response time for those systems which must support access to very large databases.

Hopefully, it will be universally accepted that piecemeal attempts to meet these goals are doomed to failure -- witness the unsatisfactory outcome of attempts to 'graft' privacy protection schemes onto existing operating systems. Consequently, successful system implementation will demand that these criteria and objectives be applied to new systems, not only from the earliest design phases, but also uniformly across all system components.

Imposition of the uniform input-output interface is one approach to achieving some of these design objectives for all system components requiring access to secondary storage. The following discussion will concentrate upon a feasible input-output module implementation, which is also compatible with the design objectives -- the proposed solution is to place most of the input-output module software in an autonomous secondary storage processor, not unlike the database processors introduced in Section 2.7.3.

5.2 Access Control and Security

The secondary storage processor enhances the input-output module's centralized access control which was discussed in Section 4.1.4. Since the necessary controlled entry point mechanism is provided by the physical partition between the processors, all process requests for input-output operations must be made via the secondary storage processor, and thus the module's access control mechanisms cannot be by-passed.

When this controlled access to input-output resources is combined with the fact that the input-output module is not forced to share a processor or, more importantly, main store with other software modules, it becomes evident that unauthorized access cannot be gained by self-modifying channel programs, or 'trapping' the central processor in supervisor mode, or tampering with operating system tables resident in main store, etc. These approaches have all met with considerable success in attempts to compromise the security of conventional systems (Linde, 1975).

In addition to the controlled access, the secondary storage devices are not 'visible' from the central processor, therefore neither systems' nor users' software can reference the devices directly. In fact, the secondary storage processor supports access to a logical structure, not a device, therefore a program may only access those physical areas onto which a valid logical structure is mapped.

However, it is necessary to by-pass the input-output module's software interface and hence the access control mechanisms for certain functions. These functions are listed below, and although executed infrequently, they <u>cannot</u> be initiated from the central processor using the uniform interface.

(1) Reconfiguration of the input-output module's internal structure, involving changes to either software, existing schema definitions, access control information, device descriptions, etc. These tasks could be executed via a special interface DML operation, however security considerations dictate that their execution cannot be initiated by <u>any</u> routine executing on the central processor.

(2) Recovery procedures following a fatal system failure may

require direct access to the secondary storage devices, without any intervening logical-to-physical translation. Again this facility should be invisible from the central processor.

(3) During Initial Program Load (IPL) or system start-up, it is unlikely that the 'bootstrap' program would be large enough to include the central processor resident portion of the input-output module. These routines would be required if the IPL was to be initiated from the central processor, via the secondary storage processor (this assumes the IPL file is in fact secondary storage resident, and not loaded fr[illegible] a special, dedicated device, e.g. a cartridg[illegible] tap[illegible] reader). Therefore, it is proposed that the secon[illegible]ary s[illegible]orage processor be started first, and then the cent[illegible] process[illegible] IPL be intiated from the secondary storage processor, using a predefined absolute main store address as the target buffer location. Once running, the central operating system could respond with an 'acknowledge' signal, to indicate that the IPL had succeeded.

Support for all these 'ultra-privileged' operations could be provided via the operator's console on the secondary storage processor. Thus, invocation of these operations would be restricted to human interaction (i.e. not software initiated) via a console which provides good physical security (i.e. a person must be in the computer room to use these facil[illegible]es). Typically, use of this access mode would be restricted to the 'database administrator', the operator, the maintenance engineers, and the systems programmers responsible for the input-output module software.

5.3 Reliability

Clearly, the system is vulnerable to total failure following either a software failure in the input-output module, or a hardware failure in the secondary storage processor. Whilst this is not necessarily worse than current systems, or the alternative designs, it is clearly unacceptable! Special precautions will have to be taken to provide an acceptable robustness to transient run-time failures.

Some of the possible fault-tolerant software techniques were discus[illegible] in Section 4.1.4, along with the improved capacity for adaptive resource utilization during unbalanced access patterns, or following a device failure. The implemented fault tolerant hardware designs will depend critically upon the degree to which extra expenditure can be justified in relation to the improved reliability. Avizienis (1976) has described three aspects of tolerance to hardware module failure, namely, identification and characterization of the operational fault, selection and use of a suitable redundancy technique, and ongoing modelling of the system's availability, reliability, fault frequency and survivability.

As an example of a fault identification technique, the inter-processor communication protocol could include automatic bidirectional consistency checks, thereby greatly enhancing the cap[illegible] for early detection of processor malfunction.

Besides programmed roll-back and retry, the designed redundancy is likely to be based upon 'macroscopic component redundancy' (Withington, 1976). Possible choices include dual data paths to all devices (i.e. switchable channels and device controllers), a multiple secondary storage processor architecture, redundant service systems (e.g. cooling and power supply), easy modular replacement of faulty units, data buffering in the device and channel controllers, a dual bus arrangement for inter-processor communication, etc. (Katzman, 1977).

Recovery from a detected failure could also be enhanced by providing some functional redundancy within the input-output module; e.g. at least two logical access paths to each data item, or parallel audit trail and logging procedures in the portions of the input-output module resident in both the secondary storage and central processors.

Obviously, any successful attempt at modelling the system's reliability will require the collection of statistics concerning the input-output module's actual operation. This facility should form part of an integrated self-monitoring process within the module, to collect statistics for use by the hardware engineers, operations staff, systems programmers, and database adminstrator(s).

5.4 Functional Dependencies Between the Input-Output Module and the Operating System

In Section 4.1.2, some comments were made concerning the software structure within the input-output support subsystems. The objective of the following discussion is to broaden the scope of those earlier comments to include the global system structure as it relates to secondary storage input-output operations, and its influence upon the global design objectives.

In attempting to illustrate the 'structure' of a moderately complex computer system, there are many criteria which may be used to decompose the 'whole' into smaller 'units' and to define the relationships between the 'units'. Possible criteria include process interaction, resource allocation and ownership, protection, design methodology, software modularization, and virtual machine abstractions. There is no a priori reason to believe that the structural models derived from such diverse criteria should display any marked similarity -- however, Parnas (1974) has shown that some similarities have historically existed.

For the purposes of the current discussion the criteria for structural decomposition has been loosely defined as <u>functional dependencies</u> between software modules. Once the modules have been identified, these functional dependencies define a <u>conceptual system structure</u> which may be used to identify relationships of the form, "module A provides services necessary for the successful execution of module B", or "when a user process requests an input-output operation modules P and Q must be invoked".

Based upon a macroscopic partition of the system software into three large modules -- a <u>user process</u>, the secondary storage <u>input-output module</u> as 'viewed' from the user process, and an <u>operating system module</u> which includes the operating system routines and application programs which are traditionally associated with the operating system, but do not require supervisor mode processing -- three conceptual structures will be considered as representative of the range of possible structures (see Figures 5.1, 5.2, and 5.3). The three structures are similar in as much as the user process is directly dependent upon both the operating system and the input-output module, however the extent of the dependency may vary from one structure to another, or even between systems with the same gross structure. Differences in the relationship between the input-output module and the operating system highlight the major variations across these alternative structures.

In arriving at these prototype structures, and their associated inter-module dependencies, it has been assumed that the processor scheduling function is not under the operating system module's juristiction. This is a reasonable approach if processor scheduling is supported by some type of 'nucleus' outside the operating system module, upon which all software modules have an intimate functional dependency.

Most current systems exhibit conceptual structures similar to the one shown in Figure 5.1, in which the execution of a user's input-output operation requires the invocation of operating system procedures. IBM's IMS/VS database management system (IBM, 1974) bears this type of relationship to its parent operating system, OS/VS. Potentially, there is a great variety in the nature of the dependency -- the operating system module may be required to initiate input-output operations, support various file access techniques, and/or perform secondary storage space allocation and management.

Some of the problems related to the adoption of this type of system structure are:

(1) Poor potection and security enforcement; refer to Section 4.1.4.

(2) Ill-defined functional and control dependencies between system components are very common, and very difficult to isolate. Consequently, independent development and

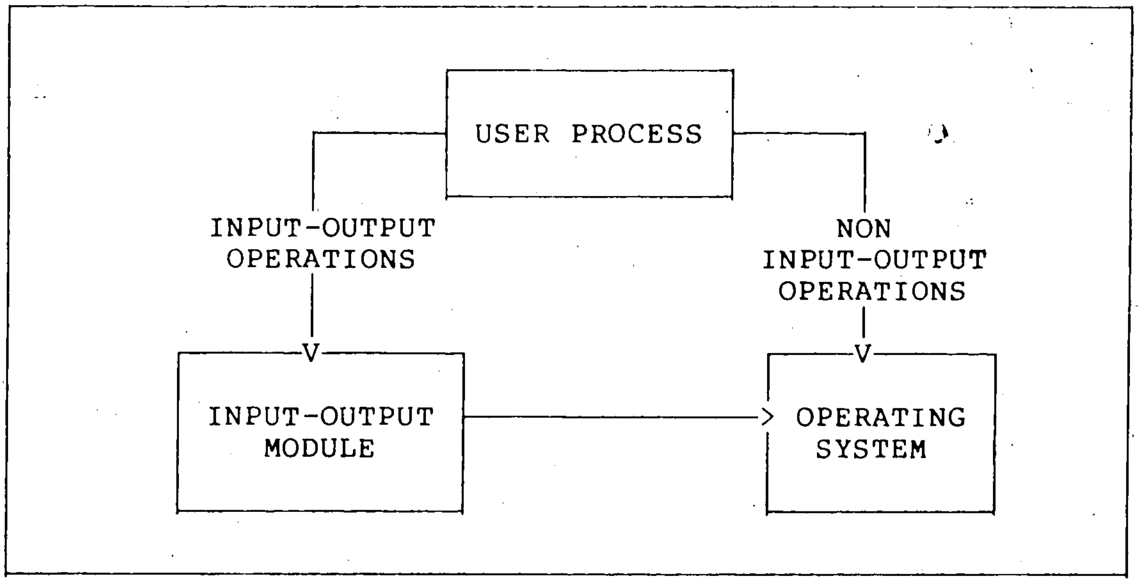

Figure 5.1 The Conventional Relationship Between the Operating System, a User Process, and the Input-Output Module

maintenance of the input-output and operating system modules is often severely hampered. Most of these problems stem from either allowing procedures to have default access to data areas and/or data structures which are unrelated to their own successful execution, or the use of module interfaces which are so weakly enforced that violations of the protocol cannot be detected.

(3) For non-trivial input-output operations (e.g. those invoked via a complex logical interface), these systems are very inefficient, due mainly to handling of the request by many layers of central processor resident software.

(4) Even assuming that the software was constructed with a high degree of modularity and module interfaces which

were both well defined and rigidly enforced, systems with this type of gross structure are not well suited to a distributed processor implementation. The tight coupling between the operating system and input-output modules implies that parallel execution could only be realized with an unacceptably high volume of inter-process communication and co-ordination.

In Figure 5.2, the input-output module has been partitioned into two submodules, one of which is completely independent of the operating system, while the other retains its dependency upon the operating system. This arrangement is typical of the backend processor approach to database management, where the database management system executes independently upon a separate processor. However, under this scheme non-database input-output is performed using conventional support routines which execute, along with the operating system, on the central processor.

Systems with this general structure display many of the disadvantages of the first organization, along with some of the advantages of the third organization, to be discussed in the following paragraphs. Consequently, no specific comments will be made about this second structural arrangement, other than the observation that support for the (non-database) secondary storage devices implies that all the associated low level input-output services, device-level protection mechanisms, directory searching procedures, and algorithms for generating secondary storage addresses from logical addresses must be duplicated in both input-output modules.

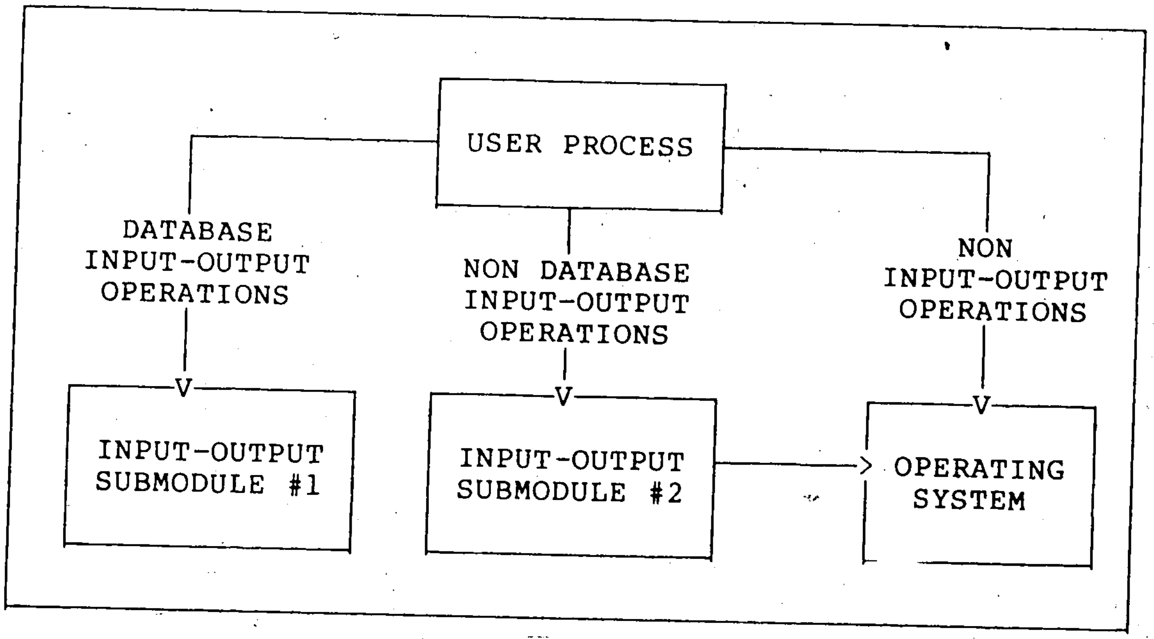
USER PROCESS
DATABASE INPUT-OUTPUT OPERATIONS
NON DATABASE INPUT-OUTPUT OPERATIONS
NON INPUT-OUTPUT OPERATIONS
INPUT-OUTPUT SUBMODULE #1
INPUT-OUTPUT SUBMODULE #2
OPERATING SYSTEM

Figure 5.2 Functional Separation Between Part of the Input-Output Module and the Operating System

System structures of the form shown in Figure 5.3 result from the adoption of a uniform input-output interface and supporting the interface from entirely <u>within</u> the input-output module (i.e. with <u>no</u> functional dependency upon the operating system). Consequently, the <u>operating system</u> becomes <u>dependent upon</u> the <u>input-output module</u>, since no input-output related functions are supported within the operating system module.

Removing all functional dependencies of the input-

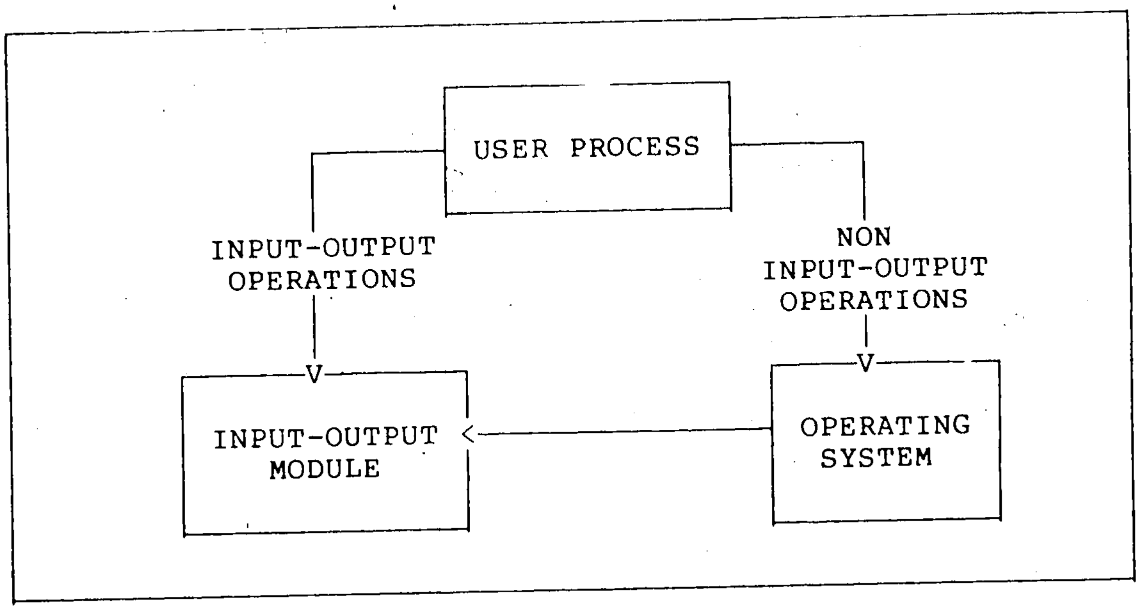

Figure 5.3 The Structure Imposed by the Uniform Interface and Its Secondary Storage Processor Implementation

output module upon the operating system does have the following apparent benefits:

(1) The enforcement of security policies is significantly simpler -- the objects requiring protection by the operating system and the input-output module have been separated and placed under unambiguous control.

(2) The obvious single functional dependency, coupled with a uniform interface permits software development and maintenance to be performed with less chance of introducing spurious secondary consequences.

(3) Functional redundancy is minimized, yielding smaller software modules.

(4) Application of a distributed processor architecture is straightforward and provides an appealing symmetry between the system's <u>conceptual</u> and <u>hardware</u> structures.

It must be stressed that the technology and expertise required to implement distributed processor configurations is available (Anderson and Jensen, 1975; Jensen, Thurber and Schneider, 1975). However, the wider acceptance of this approach to hardware organization appears to be stalled, pending the development of software structures with suitable inter-process communication protocols, which are capable of exploiting the inherent parallelism and local autonomy (Jensen, 1975; Flynn, 1977) -- the uniform input-output interface forms the basis of one such software organization.

5.5 Resource Utilization

Many of the comments in this and the following Section are prefaced by the assumptions that:

(1) The secondary storage processor will be smaller than the central processor.

(2) The cost per byte of main store attached to the secondary storage processor is significantly less than the cost per byte of central main store.

(3) For the processing tasks associated with the input-output module, the secondary storage processor provides an <u>acceptable</u> throughput for <u>less</u> cost than performing the same tasks with acceptable throughput on the central processor. Here 'cost' includes the initial capital purchase price, recurrent expenditure to operate the machine, and maintenance charges.

Recent studies, surveys and predictions appear to substantiate these assumptions, namely:

(1) Backend processors based upon minicomputers have been implemented, and shown to be capable of supporting large portions of a sophisticated database management system (Canaday, Harrison, Ivir, Ryder and Wehr, 1974; Heacox, Cosloy and Cohen, 1975; Maryanski, Fisher amd Wallentine, 1975).

(2) Minicomputer main storage is clearly cheaper (per byte) than main store for a large central processor; for example, half a megabyte of Intel Corporation's 'add-on' memory is currently listed at $4[illegible],000 for an IBM 370/158 processor, but only $20,000 for a PDP 11 series processor.

(3) The ratio of cost to processing effectiveness has been shown to favor the small processor over its larger counterpart (Juliussen and Bhandarkar, 1975). This differential in processor performance appears to reflect the more advanced technology and manufacturing techniques which can be readily incorporated into new mincomputers, as a result of their relatively short design cycles.

The functionally dedicated, distributed processor approach is economically attractive since, as Flynn (1977) points out;

"Whenever, or wherever, there is a recurring computational function that can be satisfied completely by a mini or micro computer it is now almost invariably better to isolate the function and assign it to that specific piece of than to centralize the computation on a larger and hitherto presumably more efficient engine."

Further cost-performance improvements for the secondary storage processor could be expected following specialized design and development, or emulation, of a machine tailored for the particular needs of input-output module software (refer to Chapter 6), and high volume production of this processor.

Besides the improved device utilization associated with a centralized input-output module (see Section 4.1.3), the secondary storage processor provides ~~considerable~~ scope for making efficient use of the available main store and processor resources. Studies of backend database management system (Canaday et al, 1975; Maryanski, Fisher and Wallentine, 1976) have identified the following advantages related to resource utilization (by simple analogy, these benefits are also available in the proposed secondary storage processor implementation):

(1) Less central main store allocated to the routines within the input-output module.

(2) Less central main store required per process using the input-output module facilities.

(3) A decrease in the operating system overhead associated with central processor resident routines.

(4) Less interrupt processing per request to the input-

output module.

(5) More central processor time available for normal user process execution.

Management of the storage hierarchy may be per[illegible]ed very efficiently under the secondary storage proce[illegible]s control, since:

(1) Data migration does not involve the central processor or central main store. The processor and buffer requirements for all inter-device transfers are available in the secondary storage processor and/or the mass storage subsystem.

(2) Migration for optimization (i.e. not associated with an immediate user request for data) may be performed during times in which the necessary input-output subsystem modules are operational, but idle -- this means that adaptive performance tuning may be provided for very little additional cost.

For systems featuring multiple central processors, the uniform interface and secondary sto[illegible] processor combine to provide a flexible input-output capability between any processor and any secondary storage device. Software multiplexing of the devices is more adaptive and possibly more efficient with respect to 'device connection' overheads, load levelling, and the complexity of the necessary hardware switch(es), than the alternative multi-channel controller and the lock-out mechanism necessary for hardware multiplexing.

The la[illegible] 'resource' which is put to more productive use under the proposed regime is by far the most important, namely people! As the expenditure on software development and maintenance approaches 60-80% of the total system costs (Boehm, 1973 and 1976), any improvement in programmer morale, programmer productivity, software correctness, software reliability or software portability will have a significant impact upon the total cost / benefit analysis of a computer installation. Under the proposed input-output support organization, all five fac[illegible] are improved, since:

(1) Programmers are freed from lo[illegible] input-output considerations. This is especially important in those applications where the mechanics of in[illegible]t-output operations merely act as a diversion from the central logic and organization of the code.

(2) The uniform interface clears the way for wider acceptance of higher level programming languages featuring input-output intrinsic functions which are compatible with the programming languages' data structures and procedural foundations.

(3) Software reliability is improved, and correctness enhanced through smaller, less complex software modules.

(4) Once implemented, the input-output module may be duplicated and interfaced to different central processors and different operating systems without modifying the routines which execute on the secondary

storage processor.

5.6 Scope for Performance Improvement

The backend approach to database management has provided ample evidence that the use of a dedicated external processor can provide dramatic improvements in the response-time and throughput of a database management system (Heacox, Cosloy and Cohen, 1975; Lowenthal 1976a, 1976b; Maryanski Fisher and Wallentine, 1975; Maryanski and Wallentine 1976). It is expected that a secondary storage processor implementation of the input-output module would produce similar improvements over an alternative central processor based implementation.

Simulation studies by Maryanski (1977) have indicated that the use of multiple backend processors provides a reasonable strategy for improving the throughput capacity of a database management system under conditions of "moderate to heavy input-output activity" (defined, somewhat arbitrarily, as less than 130 milliseconds of central processor execution between successive database requests for each active process). The same results should hold for a multiple secondary storage processor configuration.

One added advantage of the proposed implementation is that all central processor resident processes requiring input-output module service would benefit rom a multiple secondary storage processor configuration, as opposed to the backend database management system scheme, in which only database applications would receive improved input-output service.

In the same manner that the uniform interface permits the internal details of the input-output module's operation to be hidden from central processor resident routines, the implementor of the input-output subsystem configuration may exploit the ambiguity of the secondary storage devices to install high perfor ance search engines, or other non-standard devices, to [illegible] response time and/or throughput. These hardware changes would have a minimal impact upon the software, since a modification to the appropriate storage level schema may be all that is required. In the worst case, the input-output module code would have to be modified to add the necessary driver and support for the new device, and the resource management algorithms would have to be changed to permit data migration to / from the new device, but no user or operating system routine would have to be modified.

Over an extended period, the proposed organization provides considerable advantages for incremental system upgrades, with minimal cost and disturbance to operational software.[a] Firstly, the secondary storage processor could be introduced to upgrade the total throughput, without purchasing a new central processor (assuming the input-output module already supported the uniform interface).

Thereafter, the system throughput could be improved by any of the following actions:

(1) .pgrade the central processor,

(? add another secondary storage processor, or upgrade the existing secondary storage processor(s).

In each case, the equipment cost is less than the alternative central processor upgrade, and the associated software changes would involve either a new operating system, and hence a new secondary storage processor interface, or a new input-output module, or no change at all in the case where the replacement processor was 'object code compatible'. Any of these alternatives seems preferable to the choice offered most current installations -- a new central processor, maybe a new operating system and possibly a new set of input-output support routines.

Specific questions concerning the internal designs for the secondary storage processor and input-output module which would be required to achieve the necessary throughput and response time will be addressed in Sections 6.2, 6.3 and 6.6.

5.7 Shared Databases and Network Environments

A database is 'shared' if the same logical data is accessible from two or more equivalent database management systems. This is the logical equivalent of the physical resource sharing discussed in Section 5.5. The database management systems may execute on the same processor, on two or more homogeneous processors, or on heterogeneous processors.

Database sharing is a more realistic proposition if the database management systems are all implemented on top of the the same uniform input-output interface. Many of the associated data reformatting and translation problems vanish, since the physical data storage is controlled by the one module. Under these conditions, the input-output module acts as a common database kernel (Rosenthal, 1977). If the kernel is to be accessible from multiple central processors, then the use of a dedicated secondary storage processor is further vindicated.

For programs requiring access to information stored at a remote site in a network environment, the possible options are to explicitly establish a link between two central processors, or between the local central processor and a remote secondary storage processor. If the second alternative was possible, then a third option would also be available -- since the secondary storage processor already has a network connection, an implicit link could be established between distributed secondary storage processors. By making the <u>scope</u> of the uniform interface extend beyond the resources under the local input-output module's control, user software could be insulated from network operations. At this stage, it would be unwise to speculate on the potential use, or ramifications, of 'network-wide' data operations, however if they were justified, the secondary storage processor in conjunction with a uniform input-output interface could readily support the facility.

5.8 Distribution of Input-Output Module Software Between the Central and Secondary Storage Processors

Outside the internal organization of the routines executing on the secondary storage processor (to be discussed in Section 6.2), the only unresolved issue is how much of the input-output module code should be off-loaded to the secondary storage processor (or conversely, how much should remain within the central processor). Intuitive arguments seem to suggest that the more processing off-loaded from the central processor, the better, however the same conclusion may be reached from an alternative argument, as follows.

Initially, assume the software drivers for the secondary storage devices reside in the external processor, along with the necessary device buffers. Now consider the following input-output module components in turn:

(1) the device level schema,

(2) the function for mapping interface schema objects onto device level schema objects,

(3) the interface schemata,

(4) the function for mapping interface schema names onto unique names for the protection precedures,

(5) the concurrent access and security mechanisms,

(6) the physical access methods, and

(7) the DML routines.

It is clear that no sensible, proper subset of these components may be placed in the secondary storage processor without causing a high volume of 'housekeeping' information to be transferred between the two processors. Consequently, if the secondary storage processor is to provide a non-trivial service, then all seven components must moved out of the central processor.

The only input-output module functions which would remain to be supported by central processor resident software are those routines required for:

(1) driving the inter-processor communications protocol (to be discussed in Section 6.4),

(2) managing the User Work Areas, and

(3) identifying to which processor an interface operation should be directed (for a multiple secondary storage processor configuration).

5.9 Potential Disadvantages

Obviously, the structure proposed in this thesis is not without some inherent disadvantages -- otherwise systems would already have been constructed along the proposed guidelines! These disadvantages are almost identical to those which have been identified for the backend approach to database management. Despite the unexplained abandonment of the early backend projects (i.e. XDMS, DDM, ECAM, etc.) there is renewed interest in the concept, and the ongoing developments at MRI Systems Incorporated, Kansas State and Ohio State Universities seem to indicate that the advantages outweigh the disadvantages.

Briefly, the disadvantages (as succinctly described by Lowenthal (1976b)) are:

(1) Multiple Vendor Maintenance: the system contains at least two heterogeneous processors, connected via a link which may be 'non-standard' -- this may cause some difficulties concerning maintenance responsibility. The emergence of third-party maintance contractors (i.e. not the original equipment supplier) may provide extra freedom for the purchaser in this area.

(2) Additional Cost: the secondary storage processor involves a visible, additional cost. Unfortunately, the financial savings constitute only a portion of the benefits and even so, their impact may be less obvious (e.g. improved software longevity, or delayed peripheral

upgrade due to improved resource utilization).

(3) Reliability: viable solutions do exist, refer to Sections 4.1.4 and 5.3.

(4) Performance: it remains to be demonstrated that the secondary storage processor can provide the throughput and response time required for high perfomance applications (e.g. the paging function) -- this issue will be covered in Section 6.6. Experience with backend database management systems suggests that the requirements of the update and simple retrieval class of applications can be easily met with the proposed organization. Those applications which require extensive secondary storage searching before returning one, or a few, qualifying records will undoubtedly require special hardware assistance, since traditional software execution of these functions is becoming increasingly inadequate. However this requirement can be accommodated, and the problem is independent of the proposed input-output module organization.

CHAPTER 6

THE ARCHITECTURE OF A SECONDARY STORAGE PROCESSOR

Implementation of the proposed input-output support organization requires detailed consideration of a number of design factors which are unique to the secondary storage processor itself. These factors include the processor architecture, the inter-processor link, buffer management, and the organization of the software resident in the secondary storage processor.

Some of the difficulties associated with quantifying the predicted performance of the input-output module will be discussed, along with the methods which have been tried thus far. Consequently, the design decisions which impact performance will be made on the basis of qualitative arguments, or clear trends which have been established using the available performance estimation tools, and have an underlying intuitive appeal.

Throughout this Chapter, the term 'processor' will be assumed to refer to the secondary storage processor, unless explicitly stated to the contrary.

6.1 Performance Estimation

6.1.1 Throughput, Response Time and Resource Utilization

Since no implementation of a secondary storage processor is available for performance measurements, some preliminary performance estimation tools must be developed.

For the backend configuration, Maryanski and Wallentine have developed a discrete event simulation model as part of the project at Kansas State University (Maryanski and Wallentine, 1976; Maryanski, 1977). Unfortunately, the published results of this work are not sufficiently detailed to allow specific inferences to be drawn -- general trends are evident, but that is all. At this stage, the operation of the model is not clear, and some of the critical parameters are ambiguously defined, and/or essential parameter values are not specified.

As an alternative approach, some preliminary work has been done in the current research towards developing an analytic queueing model for a subsystem comprising a central processor, a communications link, a secondary storage processor, and multiple input-output devices. Some immediate problems were encountered in this work, namely: (1) The number of parameters is potentially very large, due to the complexity of the network which has to be modelled.

(2) It is not possible to use observed values for the parameters, and it appears that the best intentioned estimates would only be accurate to within 'half an order of magnitude'.

(3) Analysis of the resulting network is sufficiently complex that no immediate solution is obvious, and even if a solution can be found, the effort required to complete the analysis is not justified in terms of the underlying uncertainty of the parameter estimates.

Consequently, the model was heavily modified to reduce the number of nodes and parameters, under the following assumptions:

(1) At each node, the arrival of requests is described by a Poisson process, and the service times at the nodes have a negative exponential distribution.

(2) There is no request contention or interference between the actual devices. Consequently, the nodes associated with the devices section of the model correspond to identical, unique channels which operate independently. Further, all channel nodes are accessed with equal probability, and any serial correlation in the access

1: In this context, a 'network' refers to a queueing network -- a collection of connected nodes and queues, with input-output requests cycling between the server nodes -- not a distributed computer network.

pattern generated by a single DML operation is ignored.

(3) All scheduling is first come, first served (FCFS).

(4) Each DML operation requires service from the secondary storage processor once initially, and then once again following each device access.

(5) Processor service times are drawn from a single negative exponential distribution, independent of the degree to which a DML operation is nearing completion.

(6) Following each 'slice' of processor execution, the DML operation which has just been processed is either completed (i.e. ready for the result to be transmitted back to the central processor), or the DML operation requires a further input-output operation. This behaviour is modelled by assigning a static probability of completion to all DML operations which have just finished processor service -- in effect this probability defines the mean number of physical input-output operations required per DML operation.

The first assumption is critical, since it permits the well known M/M/1 queueing model[2] to be used at each node in the network. Although the M/M/1 model is not particularly realistic in this environment, it does have the important property that many M/M/1 models may be combined either in parallel, or serially, to provide analytic expressions for

2: Any introductory text on queueing theory will contain an analysis of the M/M/1 model; for example, Kleinrock (1975), Chapter 3.

the entire subsystem.

Unfortunately, the M/M/1 model will tend to overestimate the variance of the inter-arrival and service times -- this will result in longer queues, and reduced throughput estimates. This trend is illustrated by the following example; Omahen (1975) has developed expressions for the mean and variance of the flow time for a configuration in which multiple devices are connected to a single block multiplexor channel, with FCFS scheduling throughout. For one particular configuration of eight uniformly accessed devices, Omahen's expressions yield a mean service time of 75.25 milliseconds, and a variance of 105 milliseconds. A negative exponential distribution with a mean of 75.25 has a variance of 5660!!

In the next Section, attempts to overcome the second assumption will be described. The remaining assumptions (i.e. (3) to (6)) are probably not unduly restrictive for a first order approximation to the system's behaviour.

Figure 6.1 illustrates the simplified model, in which DML requests enter via a queue for the central to secondary storage processor link, are transferred to the secondary storage processor and queued awaiting execution. Once served, the request either exits via the queue for the secondary storage to central processor link, or is queued for a physical input-output at one of the channel/device servers. Upon completion of a physical input-output operation, the request re-joins the single queue awaiting secondary storage processor service.

The M/M/1 queueing model has been implemented, and tested, however some further problems remain. Firstly, it has not been possible to reconcile the model's predictions with the Kansas State simulation results; for an example, refer to Figure 6.3 in Section 6.2. In particular, this relates to absolute values, not trends, and may be partly due to an incomplete description of the simulation model, and partly due to the M/M/1 model's underestimated throughput as discussed earlier. Secondly, the queueing model is rather sensitive to variations in the parameter values. From earlier experience with the stochastic simulation of file organizations, it is evident that the combination of parameter sensitivity and a large number of input parameters, each with a wide range of 'plausible' values, leads to a situation in which the model's use may be severely limited.

Consequently, the results from the queueing model will not be explicitly presented, however a few of the observed <u>general</u> <u>trends</u> will be used to substantiate the arguments and designs presented in the following Sections.

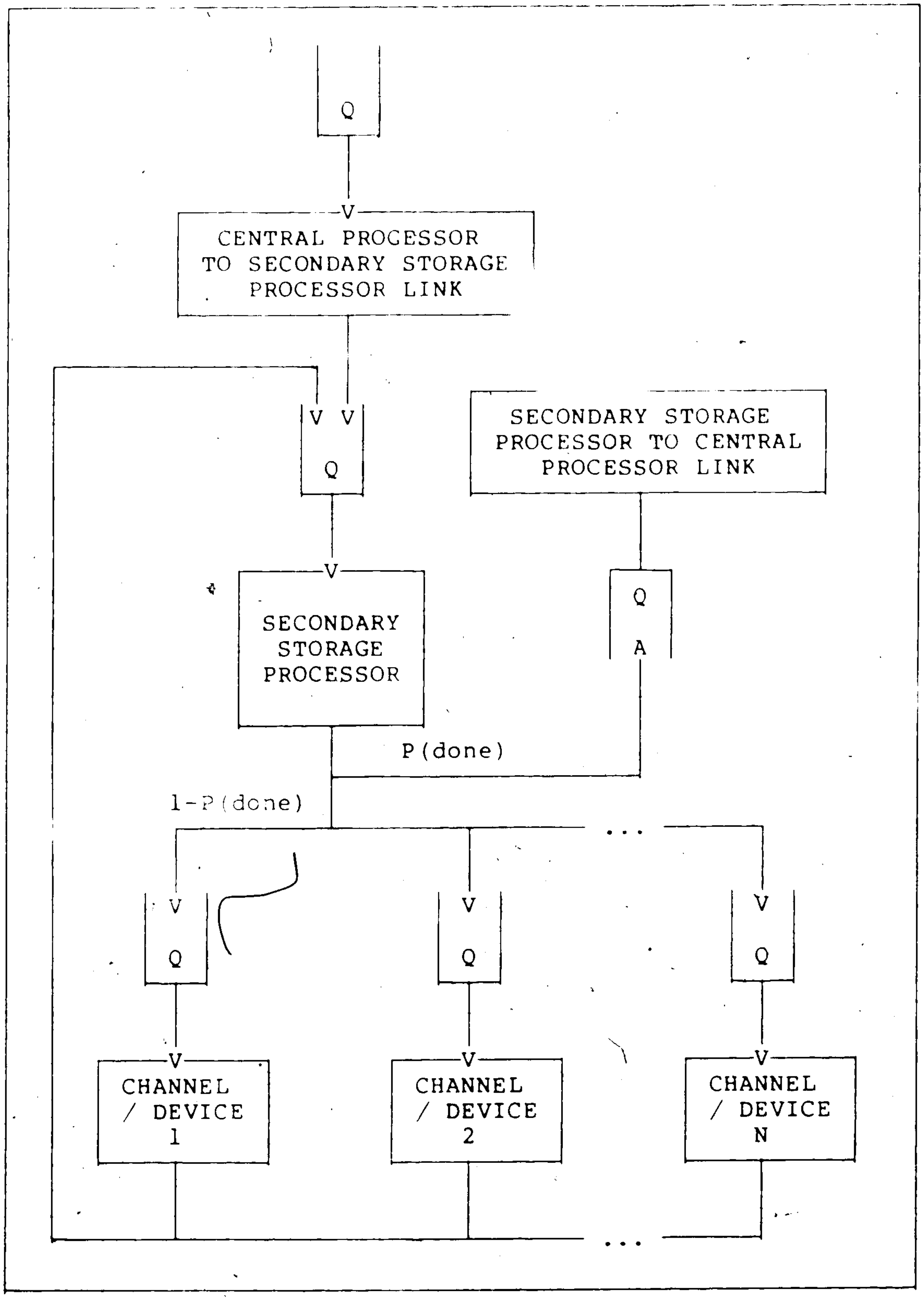
Q
CENTRAL PROCESSOR TO SECONDARY STORAGE PROCESSOR LINK
V V
Q
SECONDARY STORAGE PROCESSOR TO CENTRAL PROCESSOR LINK
SECONDARY STORAGE PROCESSOR
Q
A
P(done)
1-P(done)
V
Q
CHANNEL / DEVICE 1
CHANNEL / DEVICE 2
CHANNEL / DEVICE N

Figure 6.1 The Simplified Queuing Model for the Secondary Storage Processor Subsystem

6.1.2 Interference Between Concurrent Data Transfer Operations Within the Input-Output Subsystem

Another performance related investigation concerns the effects of concurrent input-output operations within the device, channel, and controller subsystem -- the desired predictions concern the degree to which potential transfers are 'blocked' because a channel or controller or device is not available, and the expected peak and mean transfer rates of the entire device subsystem. This information is required when estimating the device service time distribution for the throughput analysis, and when considering the necessary bandwidth for the communications link between the central and secondary storage processors.

Rosenthal (1977) has made some first order approximations in this area, however these calculations ignored the distribution of requests between devices, and the details of the path connections between shared devices, controllers and channels.

As part of the preliminary investigations for the current research, some work was done in this area of transfer concurrency. The general approach involves modelling the subsystem with a directed graph in which the nodes correspond to the secondary storage processor, the channels, the controllers, virtual controllers (added to enforce the constraint that the controller can only service one device at any instant), and the devices. Each arc corresponds to a data path, and is assigned an appropriate transfer bandwidth (i.e. a 'flow capacity'). Finally, static access probabilities must be assigned to the device nodes. Figure 6.2 shows the graph for a sample subsystem with 7 disk devices, 3 controllers and 2 channels.

The estimation procedure initially requires the enumeration of all the possible <u>access sets</u> involving 1 access request, 2 requests, 3 requests, and so on until the number of requests in the access set is equal to the number of channels connected to the processor. An access set of N members is formed by sampling N times <u>with replacement</u> from the set of device nodes in the configuration graph. For each access set, an adaptation of the well known 'maximal flow' algorithm[3] is used to determine how many of the desired accesses can be performed concurrently. Once this has been established, the access set's probability of occurrence may be determined from the individual device access probabilities, and thus the mean and maximum transfer rates can be computed.

As with the queueing model, these measurement techniques have not been rigorously pursued, because:

(1) The omnipresent parameter value problem poses further difficulties.

(2) The technique assumes heavy device utilization, and in

3: Refer to Deo (1974), Chapter 14, pp 384-393.

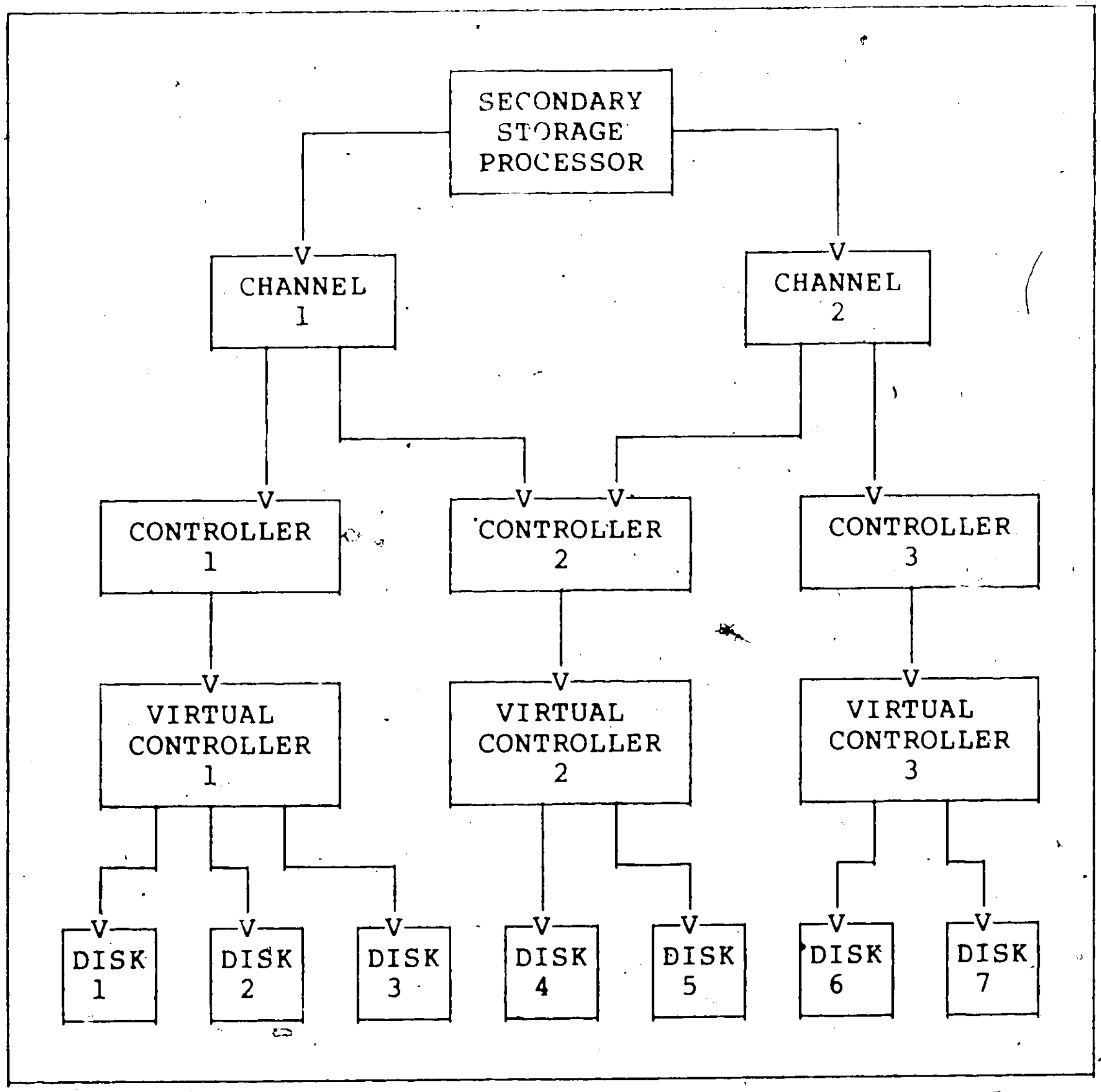

Figure 6.2 Directed Graph Representation of a Sample Device Configuration

its current form cannot be used in a situation featuring significant device idleness.

(3) Implicit in the method is the assumption that the scheduling algorithm and incremental nature of path allocation to asynchronous requests will not influence the attainable transfer concurrency -- unfortunately

this is not true. These factors would reduce the projected mean transfer rate by an undetermined amount, and there is no obvious way in which the method could be adapted to correct this omission.

(4) For a single configuration, the method has some potential (e.g. for tuning or reconfiguring a particular implementation), however, other than trivial results, no valid generalizations have been found.

Thus, despite good intentions, it appears that considerable effort (beyond the scope of this research, and current technical skills of this researcher!) must be invested to produce adequate performance estimation tools.

6.2 Request Multi-Tasking

Undoubtedly, the secondary storage processor will have to support multi-tasking between simultaneous DML operations. This claim is based upon the performance estimates from the Kansas State simulation studies and the M/M/1 queueing model, which both indicate that the improvement in throughput achieved by allowing the secondary storage processor to handle 2 or more DML operations concurrently is so great, compared to serial processing, that is cannot be ignored. Basically, the rationale is simply that the secondary storage processor should have something useful to do during physical input-output.

Some sample estimates, representative of the general trends, are presented in Figure 6.3, based upon secondary storage processor throughput for varying levels of multi-tasking. Two operational environments have been modelled; the first (Experiment 1) is taken from the Kansas State study, and illustrates an implementation which features no communications overhead, and a small number of physical input-output operations per DML operation. In Expermient 2, the DML operations generate more physical input-output, and some communications overhead has been included. The critical parameter values are presented in Table 6.1.

Once the decision has been reached to provide some multi-tasking facilities, choosing the limit on the maximum number of operations to be handled concurrently depends upon the particular application. Significant factors include the amount of local main store available for interface schemata and buffers, the required response-time, the projected processor utilization, and the desire to maintain the processor task switch overhead within reasonable limits. It should be noted that the relative improvement in throughput associated with adding an additional task decreases with the number of tasks. Therefore, the largest increase occurs at the transition from serial processing to concurrent processing of two DML operations. The software complexity required to support multi-tasking is virtually all associated with that same initial step from one to two concurrent tasks -- after that, higher levels of multi-tasking may be supported without a significant increase in

Parameter	Experiment 1	Experiment 2
Mean Number of Physical Input-Output Operations per DML Operation	0.443	1.5
Number of Parallel Channel/ Device Modules	8	8
Mean Device Service Time per Input-Output Operation	30.0 msec	30.0 msec
Mean Link Transfer Time from the Central Processor	0.0 msec	2.0 msec
Mean Link Transfer Time to the Central Processor	0.0 msec	2.0 msec
Mean Secondary Storage Processor Service Time per DML Operation (Includes Task Switch Overhead)	14.4 msec	25.0 msec

Table 6.1 Parameter Values for the Performance Estimates of Figures 6.3, 6.4 and 6.5

software complexity.

Figures 6.4 and 6.5 have been included to illustrate the changes in response-time and processor utilization which can be typically expected as the level of multi-tasking increases. Note that the although the throughput and processor utilization appear to level off, the response-time increases linearly as more concurrent tasks are added. Therefore, while multi-tasking improves both the <u>rate</u> at which DML operations may be executed and the <u>capacity</u> of input-output module, the <u>turnaround</u> for a single operation

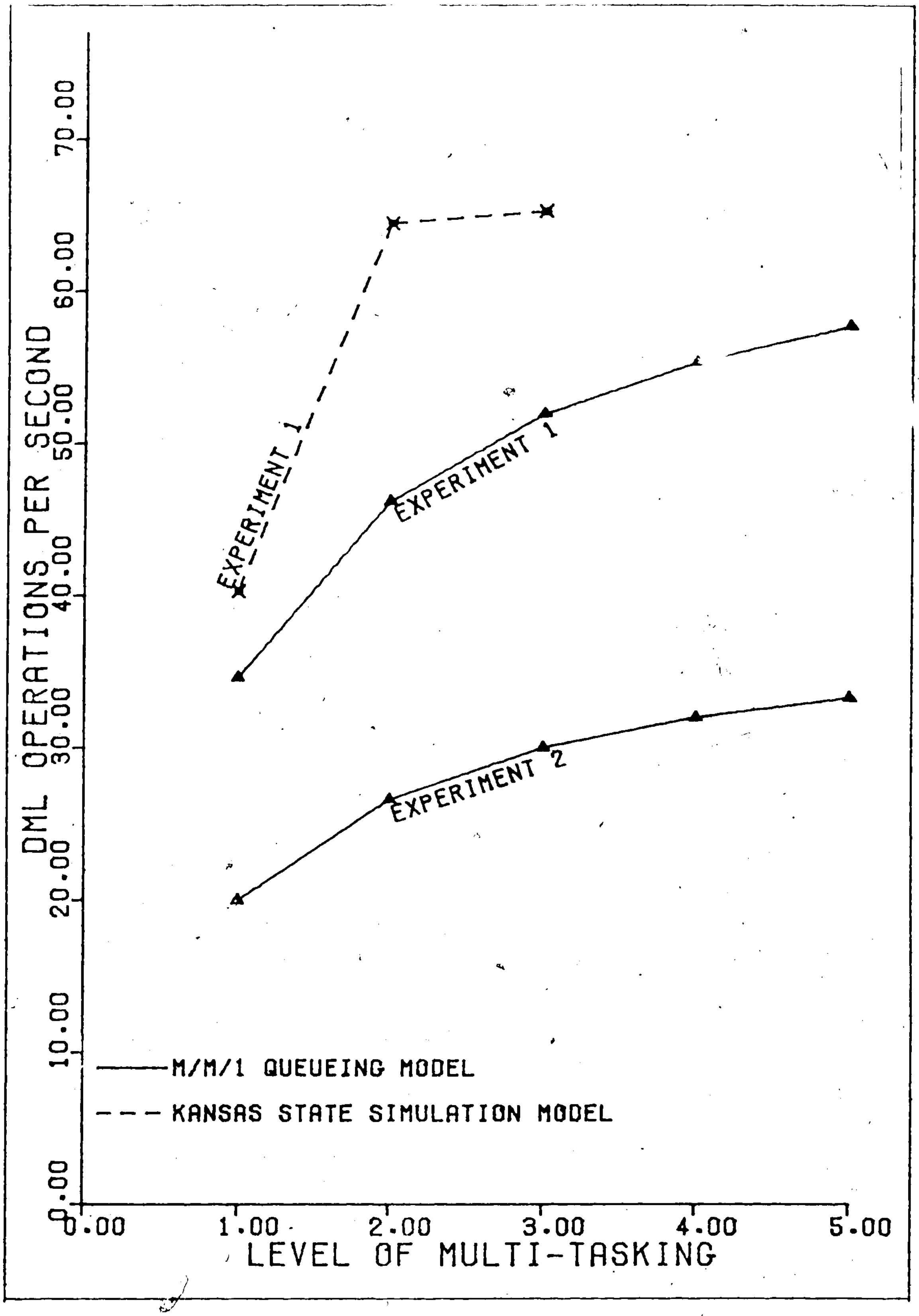

Figure 6.3 Sample Throughput Estimates

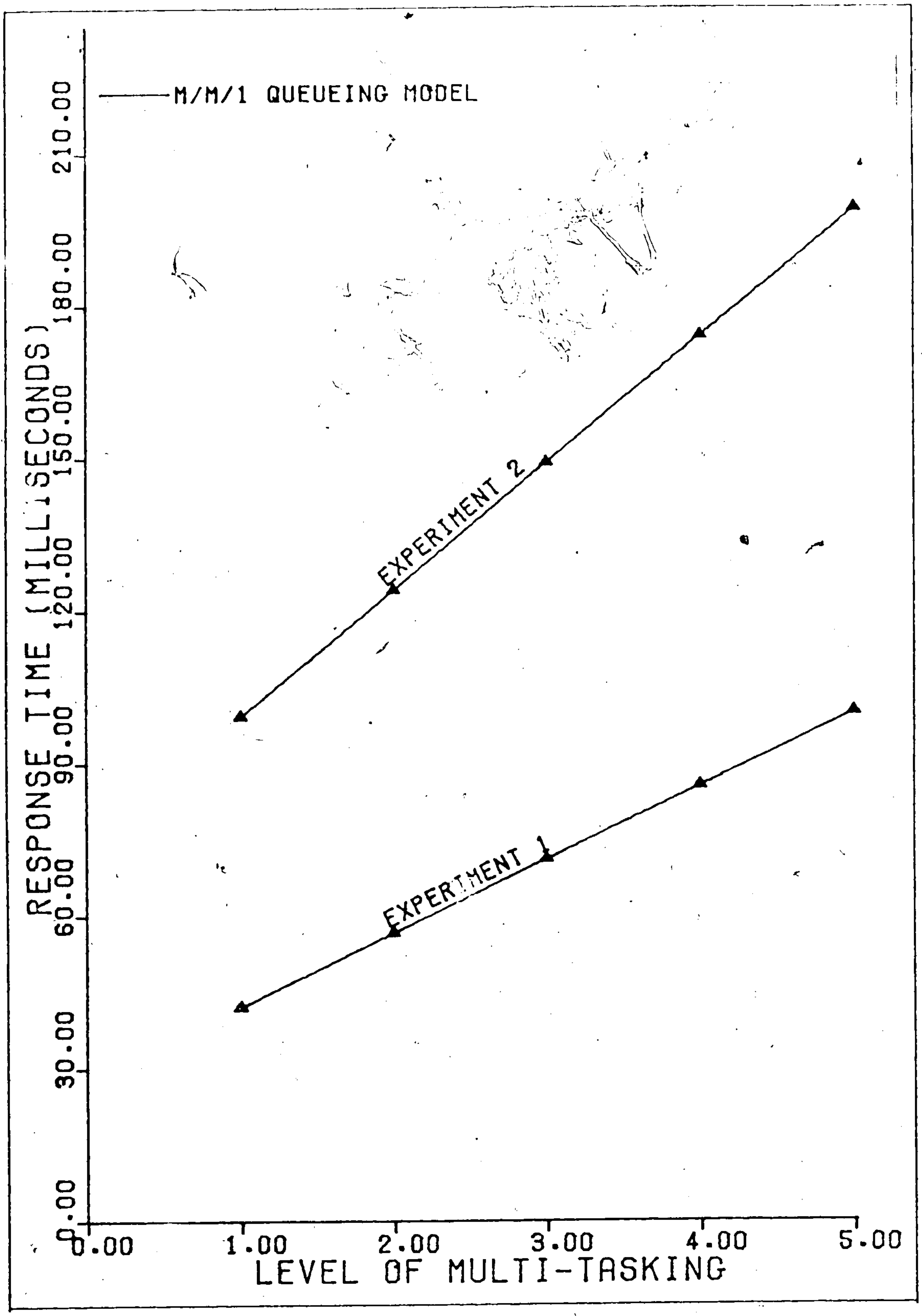

Figure 6.4 Sample Response Time Estimates

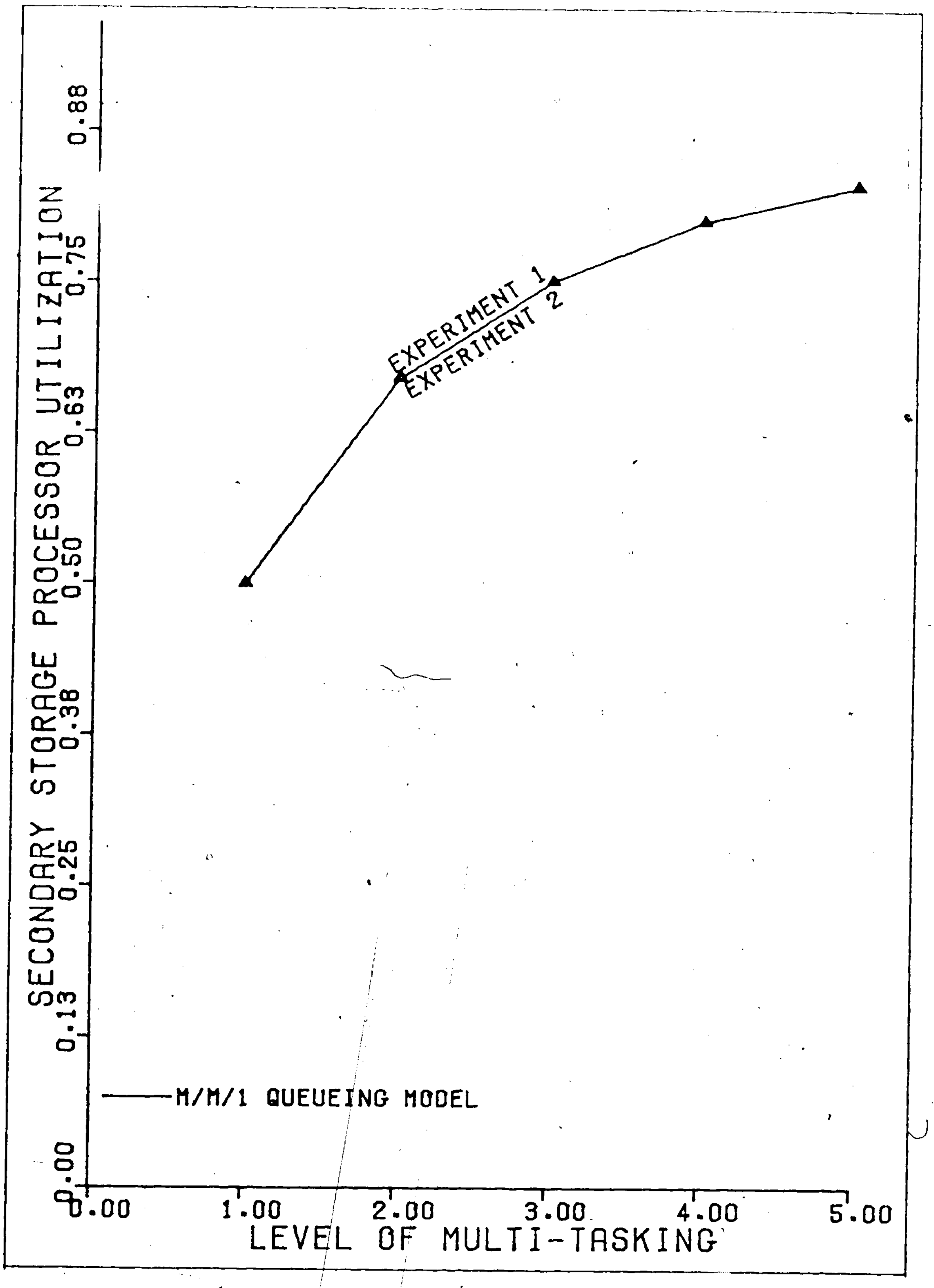

Figure 6.5 Sample Secondary Storage Processor Utilization Estimates

may deteriorate significantly, even when the throughput increase is only marginal.

Within the Kansas Sate project, multi-tasking has been approached with an "OS MFT" philosophy, in which a fixed number of partitions, or tasks, are created in the backend, and a dispatching algorithm is used to allocate an outstanding DML request to a partition when one becomes free. There is no apparent difficulty associated with an alternative operating system for the secondary storage processor which would support a level of multi-tasking which varies with the demand, up to the limit imposed by the available resources and response-time constraints.

A number of fundamental design decisions concerning the processor and its operation are impacted by the requirement for a multi-tasking environment. For example, the software overhead associated with task switching, and hardware techniques to reduce this overhead assume increased importance, a 'task' must be clearly defined (a task could be assigned to a partition of the logical data resources, to a partition of the physical resources, to one active DML operation, or to a particular function within the input-output module), efficient inter-task communications facilities are obviously required (possibly spanning physical processor boundaries), etc., etc.

It should be noted that the implicit locking mechanism mentioned in Section 4.3.2.1 is essential to the correct operation of a multi-tasked secondary storage processor. Depending upon the definition of a task, further inter-task locks may be required for logical and/or physical resources.

6.3. The Hardware Architecture of a Secondary Storage Processor

When considering the internal architecture of a secondary storage processor, a number of desirable characteristics are immediately obvious. However, beyond these basic features, further options are less clearly justified. Therefore, an effective implementation strategy would appear to involve first choosing (or designing, then building and/or emulating) a machine with the basic attributes, and then implementing the input-output module. Not until this has been done can the real requirements for the less obvious features (e.g. a 'tagged' architecture, a cache memory, or hardware implemented 'capability' mechanisms) be evaluated.

The basic attributes have been independently formulated by at least three different groups (i.e. MRI Systems Corporation, the Kansas State project, and at an early stage of the current research). There is remarkable agreement between these proposals, namely:

(1) A 32-bit minicomputer seems to form the most logical base machine, for three reasons:

(a) These machines provide sufficient raw processor speed.

(b) The total main store address space and the maximum address space for a single task on a 16-bit minicomputer is simply not large enough to hold the required software and buffer areas without partitioning and/or overlaying, which would impose an unacceptable overhead upon the processor's operation. These storage requirements for conventional database management system have been variously estimated to lie in the range of 5 x 10^4 to 1 x 10^6 characters (IBM, 1974; Wallentine, Maryanski, Fisher, McBride, Fox, Chapin and Allen, 1975). Typically, a 32-bit machine would support 20- or 24-bit internal addresses, giving a total address space of 1 - 16 x 10^6 main store locations. This is large enough to support multiple (i.e. 8 or 16) tasks with adequate address spaces in the range of 1 x 10^5 to 1 x 10^6 main store locations.

(c) Support for very large databases and/or large volumes of integrated on-line storage will require internal physical pointers which are capable of addressing of the order of 10^9 items resident in secondary storage. This implies pointers of at least 32 bits. On a machine with 16-bit data paths and arithmetic logic unit (ALU), pointer manipulations would be very expensive compared to a 32-bit machine.

(2) The requirement for a small task switch overhead has been mentioned previously. Machines with a 'multiple stack' capability appear to have a considerable advantage in this area over the 'general purpose register' based architectures.

(3) Since the bulk of the interrupt processing load has been shifted from the central processor, the secondary storage processor must support very efficient mechanisms for interrupt handling. A multiple priority level, vector based interrupt mechanism would minimize the interrupt scheduling and identification overheads. Actual processing of an interrupt may be speeded up by providing multiple general purpose register sets, or employing a stack architecture.

(4) The non-numeric nature of the work load on the secondary storage processor dictates that some hardware aids be provided for character string manipulation and comparison. This is especially true if no database search engines are employed, and the searching function is implemented in software. Obviously, local main store should be character addressable, and the data manipulation facilities should be enhanced, either with a very fast 8-bit ALU / shifter / comparator, or a character based ALU with variable length, descriptor based inputs and output(s).

(5) Communication between the secondary storage processor and the input-output devices should be achieved via

central processor compatible channel controllers and channels, with bandwidths not less than those of the channels att hed to current central processors. Typically, this will require an upgrade in the input-output interfaces provided with present minicomputers, however the investment is essential if the transfer rates and distributed processing capabilities at the channel and device controllers of current input-output subsystems are to be maintained, or extended.

(6) Reliability and/or performance considerations may dictate that a multiple secondary storage processor configuration be used initially, or at some forseeable subsequent upgrade. In this situation, the processor architecture must provide additional features for local main store sharing between processors, and inter-processor communication. In addition, the channel controllers would have to provide switchable connections to more than one processor.

6.4 Factors Influencing the Design of the Link to Connect the Central and Secondary Storage Processors

The 'link' between the central and secondary storage processors consists of some hardware, and one software routine in each processor to control the hardware. For the hardware component, three choices are available:

(1) A standard telecommunications line.

(2) A channel-to-channel adaptor.

(3) A shared bus.

Of these, the first is the cheapest, supports the lowest throughput and, although it is suited to configurations in which the central processor and secondary storage processor are not situated in close proximity, it is probably unsuitable for the application proposed here. This decision is based upon the maximum bandwidth of such a link, which would be about 50 Kbytes per second. Even with the reduced bandwidth requirements of the central processor resident software (since all the data associated with physical input-output does not have to be transferred to the central main store), this upper limit is apparently too small to cater for all input-output module operations. For example, the block multiplexor channel on a conventional system may have bandwidth of approximately 3 Mbytes per second; the reduction in data transfer rate between the devices and the inter-processor link would have to be 60:1 before a 50 Kbyte per second communications link could provide the same throughput of useful information to an applications program.

Channel-to-channel adaptors provide an immediate solution with reasonable capacity (approximately 1 - 4 Mbytes per second). These devices are currently available from minicomputer vendors, to permit their machines to be interfaced to larger central processors. Typically, a channel-to-channel adaptor emulates a standard sequential device for both the attached processors (e.g. the link may appear to be a magnetic tape unit, or a paper tape reader / punch). Herein lies one of the potential disadvantages, namely the link control software must support a device which really has nothing to do with the task of inter-processor communication. One further disadvantage of the approach is the tightly coupling which is enforced between one central processor and a particular secondary storage processor.

The shared bus communication technique would permit data transfers between the main store attached to any one of a number of central and/or secondary storage processors. This flexibility could be enhanced by an intelligent bus controller, which is capable of re-routing transfers in the event of a processor or main store module failure. Transfer rates from a common bus could conceivably approach 50 Mbytes per second in the forseeable future.

In order that the processor resources devoted to servicing the link are minimized, it is essential that either the hardware link, or the processor interface to the link should contain sufficient logic and buffer storage so that once initiated, a single transfer generates no more than one interrupt per processor.

For current systems, the channel-to-channel adaptor method is the most attractive. Since it can be incorporated into existing hardware and input-output subsystem architectures, this choice appears to give the best compromise between bandwidth potential and engineering costs associated with non-standard interfaces. The shared bus approach will only become feasible when, and if, the central processor and main store modules are redesigned around a high bandwidth central data bus, with external ports.

Given the channel-to-channel communication facility, software drivers in each processor must not only 'drive' the link, but implement the basic functions for communication between input-output module components located in separate processors. This will require buffer management, error checking on completed messages, and routing messages to the correct destination processor (a message will be addressed to a particular process, either in the in the secondary storage processor or the central processor resident portion of the input-output module).

6.5 Tasks, Buffers and Software Organization

As described in Section 5.8, the software executing on the secondary storage processor will include virtually all the components of a conventional database management system. It is expected that the processor will function under the control of a specialized operating system, whose principal functions will include processor multiplexing between tasks, inter-task communication and synchronization primitives, and enforcement of each task's local address space limits. The scheduling procedures within the input-output module's operating system must cater for tasks with varying priorities, in order that the necessary performance may be achieved, and less important tasks (e.g. internal management of the storage hierarchy) do not inhibit the quick processing of urgent DML requests. Given these basic facilities, all other functions may be implemented directly by the input-output module software, which in all likelihood would be reentrant.

It is proposed that one task should exist within the secondary storage processor for each active DML operation, where 'active' implies a message initiating the operation has been received by the secondary storage processor. This choice permits the number of tasks to vary with the demand, hence maximizing the main store resources available to the active tasks, and allows all the variable information associated with a single DML operation (e.g. tables, status information, schema, record and message buffers) to be assigned to one task.

Since the central processor resident section of the input-output module is responsible for dispatching new DML comm[illegible] this module must also assume responsibility for ensuri[illegible] the secondary storage processor is not overloa[illegible] Transmission of a new DML operation to the seconda[illegible] storage processor may be delayed if, that processor [illegible]

(1) indicates that [illegible] ma[illegible]um level of multi-tasking has

been reached, or

(2) indicates that no schema or message buffers are available, or

(3) is not operational.

6.6 Other Techniques Designed to Improve Performance

A number of options are available for performance improvements to be introduced via extended interface schema facilities. These involve 'hints' to the input-output module regarding the application environmnent, schema usage, or processing patterns. From the application's viewpoint, these extra declarations are transparent, except fo a possible reduction in the interface's capacity to support full data independence.

DDL declarations could be added to invoke some of the following functions, for a few performance sensitive applications (e.g. the paging subsystem):

(1) 'Turn off' all structural transformation for a particular record type.

(2) Permit the interface schema for exclusive, single application usage -- this allows the concurrent access checks to be executed once when the schema is OPENed, and then by-passed during subsequent accesses.

(3) The specified buffer address is in the user's data area, not the User Work Area (i.e. do not move records to / from the User Work Area).

(4) Inhibit statistics collection.

(5) Prefetch the 'next' record whenever possible.

(6) Retrieve the member records whenever the owner of a particular set type is retrieved.

(7) Allocate extra large buffers for a particular record type.

For applications requiring quick response, it is reasonable to assume that DML operations, and hence input-output module tasks, would be assigned a high priority. Further factors which contribute to faster response-times and higher throughput include:

(1) Automatic management of the secondary storage hierarchy will tend to minimize the physical access and transfer times between the devices and the secondary storage processor.

(2) It is possible for a single software module (e.g. a paging subsystem) to request many DML operations which may, in fact, be subsequently executed in parallel.

(3) Since the secondary storage processor's operating system is highly specialized, and a single input-output module is implemented, the software overhead associated with an input-output operation will likely be less than the overhead incurred during a request's passage through multiple layers of software and redundant checks in a conventional system.

Under these assumptions, the peak rate at which the input-output module could handle requests may indeed be <u>higher</u> than the maximum rate achievable with direct execution of physical input-output.

CHAPTER 7

CONCLUDING REMARKS

7.1 Summary

A study of the evolution of hardware components within the input-output subsystem has shown that, for medium-to-large scale machines, the adoption of distributed processor architectures is already well established. This trend seems likely to gain momentum over the next decade as the benefits of the approach -- increased parallelism, more cost-effective operation due to special purpose modules, and improved resource utilization and management -- are more widely recognized, and accentuated in the face of falling hardware costs.

Compared to their earlier counterparts, the use of external support processors will extend beyond management of the device control functions, as much of the repetitive, specialized input-output processing is off-loaded from the central processor. Under these circumstances, the secondary storage devices may be entirely hidden from the processes executing on the central processor, and replaced by a device independent interface to an external input-output processor, or database processor.

From the software perspective, current approaches to input-output support are generally rather unsatisfactory. Uncontrolled and incremental development of the various support subsystems has left the applications programmer with a legacy of multiple interfaces supporting functionally equivalent operations in a variety of manners. Protection enforcement is often weakest within these input-output code modules. The total performance of the system is hampered by excessive software overheads and non-adaptive resource management.

Software development and maintenance costs associated with the input-output support routines are unnecessarily high, due to the poor organization of some very complex pieces of code. This lack of software structure and obscure functional dependencies also inhibits the implementor's freedom to utilize new hardware input-output components and external support processors. Since the necessary software changes (e.g. to interface a new device or controller, or to off-load certain functions to a dedicated processor) are usually quite extensive, the potential performance improvement is devalued by the large changeover costs and unreliable operation following a major software reorganization.

Consequently, the homogeneous secondary storage input-output interface has been proposed, as a unified solution to the software problems, which may be easily integrated into either existing, or future, hardware configurations, as a result of its proposed secondary storage processor implementation.

Compared to conventional techniques, the proposed organization for secondary storage support offers easier usage from high level programming languages, smaller, more easily comprehended source programs, improved security and protection enforcement, superior throughput and response-time performance, more effective resource utilization, and greater flexibility during hardware reconfiguration. In general terms, a homogeneous interface based upon a secondary storage processor architecture provides a system which has an attractive cost-performance ratio compared to the alternative organizations.

7.2 Significance

Whilst the findings of this research are certainly encouraging, it must be stressed that the justification conducted thus far for the proposed system organization is exploratory and directed towards general questions of apparent feasibility. An attempt at detailed design, performance evaluation, and complete justification lies beyond the physical, technical and chronological constraints of the current research.

However some significant conclusions regarding design techniques may be drawn from the work completed to date. Firstly, an integrated design based upon either current, or projected, technologies, components, and attributes which are universally accepted as desirable, <u>is</u> intellectually possible -- the problem is <u>not</u> too large to justify avoiding it. Any successful attempt at implementing a system requiring sophisticated secondary storage input-output facilities must be based upon a very broad perspective of the hardware and software components, and their mutual interaction. Piecemeal approaches have been inadequate in the past, and will become even more so as the attention shifts from the hardware constraints to the global design weaknesses in present systems. Further, the design process should be initially guided by global objectives, and the finer implementation details should evolve in a general 'top-down' sequence.

In its completed form, the proposal presented in this thesis constitutes a significant departure from accepted design and implementation practices. However, the achievement of a single input-output module, running on an external support processor, does not demand the construction of a new machine or operating system from 'the ground up' -- existing hardware and software may be used to realize the completed proposal through a series of incremental evolutionary steps.

7.3 Outstanding Problems and Areas Requiring More Detailed Investigation

One of the remaining unresolved issues concerns the suitability of the proposed interface for implementing multiple database management systems. Basically, two subproblems arise, (a) "How convenient is the interface for supporting non-network data models, and implementing non-software, end-user interfaces?", and (b) "Can the input-output module be used to implement procedures which would in turn support multiple 'views' of a single logical data structure, or could this facility be provided entirely by the input-output module?". The first question concerns the extent to which the interface facilities are functionally complete. Resolving the second question is more difficult, and assumes considerable design of the input-output module's internal structure mapping and locking mechanisms as a prerequisite.

The interface's functional completeness also comes under consideration when an attempt is made to complete the design of those operating system and utility applications which require the input-output module capabilities to be substituted for the input-output facilities upon which they are currently based.

Another area of potentially fruitful research would be to assume the desirability of the symmetry between the input-output support system's conceptual (functional) and physical (hardware) decomposition, and then to investigate the applicability of generalizing this approach to other macroscopic system support functions.

Perhaps one of the most potentially productive areas warranting further research is that of performance evaluation. This work should be directed towards two goals, namely:

(1) Increased monitoring of current systems to extract reliable parameter values. Undoubtedly, '' is is already being done within the manufacturers' software support groups, however very little quantified evidence has appeared in the published literature.

(2) The current modelling and analysis techniques need to be extended to remove some of the more restrictive assumptions of the earlier attempts. Advanced concepts in queueing theory and controlled, discrete event simulation could possibly be combined to produce some hybrid model of the secondary storage processor subsystem, which is significantly more realistic than the current models.

The last major issue worthy of ongoing investigation relates to inter-processor communication -- both the hardware and software aspects. Initially an efficient mechanism must be found to handle the central processor to secondary storage processor link, first for one processor at each end, then for multiple processors. However a related problem, but one for which possible solutions are much less apparent, concerns the link between the secondary storage processor and a highly parallel associative search engine. At this stage, it is unclear where these devices fit into the overall 'scheme of things' for systems with central processor based input-output support, therefore there are no predecessor systems to study. Perhaps the secondary storage processor organization will make it easier to use these very specialized modules efficiently, since they are not visible for the central processor resident software, however the issue is not at all clear.

BIBLIOGRAPHY

Allman, Eric; Stonebraker, Michael; Held, Gerald (1976): "Embedding a Relational Data Sublanguage in a General Purpose Programming Language", Memorandum No. ERL-M564, Electronics Research Lab., Univ. of California, Berkeley, October.

Anderson, George A.; Jensen, E. Douglas (1975): "Computer Interconnection Structures: Taxonomy, Characteristics and Examples", Computing Surveys, vol 7, no 4, pp 197-213.

Anderson, George A.; Kain, Richard Y. (1976): "A Content-Addressed Memory Designed for Data Base Applications", Proc. 1976 Internat. Conf. on Parallel Processing, Wayne State U., Michigan, August, IEEE Comp. Society, California, pp 191-195.

ANSI/X3/SPARC Study Group on Data Base Management Systems (1975): "Interim Report 75-02-08", ACM SIGMOD FDT Bulletin, vol 2, no 2, pp 1-140.

Atkinson, Toby (1974): "Architecture of Series 60/Level 64", Honeywell Computer Journal, vol 8, no 2, pp 94-106.

Attanasaio, C.R.; Markstein, P.W.; Phillips, R.J. (1976): "Penetrating an Operating System: A Study of VM/370 Integrity", IBM Systems J., vol 15, no 1, pp 102-116.

Avizienis, Algirdas (1976): "Fault-Tolerant Systems", IEEE Trans. on Computers, vol C-25, no 12, pp 1304-1312.

Bachman, Charles W. (1972): "The Evolution of Storage Structures", Comm. ACM, vol 15, no 7, pp 628-634.

Bachman, Charles W. (1975): "Trends in Database Management - 1975", Proc. AFIPS National Comp. Conf., vol 44, Anaheim, California, May, AFIPS Press, Montvale, New Jersey, pp 569-576.

Balzer, Robert M. (1971): "PORTS - A Method for Dynamic Interprogram Communication and Job Control", Proc. AFIPS Spring Joint Comp. Conf., vol 38, Atlantic City, New Jersey, May, AFIPS Press, Montvale, New Jersey, pp 485-489.

Balzer, R.M. (1973): "An Overview of the ISPL Computer System Design", Comm. ACM, vol 16, no 2, pp 117-122.

Barron, E.T.; Glorioso, R.M. (1973): "A Micro Controlled Peripheral Processor", Preprints 6th Annual Workshop on Microprogramming, Maryland, September, ACM, New York, pp 122-128.

Baum, Richard Irwin (1975): "The Architectural Design of a Secure Data Base Management System", Report OSU-CISRC-TR-75-8, Computer & Information Sci. Research Center, Ohio State U., November, (NTIS report AD A021 158).

Baum, Richard I.; Hsiao, David H. (1976): "Database Computers - A Step Towards Data Utilities", IEEE Trans. on Computers, vol C-25, no 12, pp 1254-1259.

Baum, Richard I.; Hsiao, David K.; Kannan, Krishnamurthi (1976): "The Architecture of a Database Computer, Part I: Concepts and Capabilities", Report OSU-CISRC-TR-76-1, Computer & Information Science Research Center, Ohio State U., September, (NTIS report AD A034 154).

Berndt, Helmut (1974): "A Multi-microprocessor Design", Preprints 7th Annual Workshop on Microprogramming, Palo Alto, California, September, ACM, New York, pp 299-306.

Berra, P. Bruce; Singhania, Ashok K. (1976): "A Multiple Associative Memory Organization for Pipelining a Directory to a Very Large Data Base", Digest of Papers: 12th Spring COMPCON San Francisco, February, IEEE Comp. Society, Long Beach, California, pp 109-112.

Bhushan, Abhay (1972): "The File Transfer Protocol", NIC Document # 10596, Network Information Centre, Stanford Research Institute, Menlo Park, California, July.

Bilofsky, Walt; Irons, Edgar T. (1973): "PDP-10 IMP72 Version 1.5: Reference Manual", Dept. of Computer Science, Yale University, New Haven, Connecticut, August.

Blasgen, Michael W. (1975): "A Comparison of Two I/O Programming Interfaces", IBM Research Report RJ 1510, Thomas J. Watson Research Center, Yorktown Heights, New York, February.

Boehm, Barry W. (1973): "Software and Its Impact: A Quantitative Assessment", Datamation, vol 19, no 5, pp 48-59.

Boehm, Barry W. (1976): "Software Engineering", IEEE Trans. on Computers, vol C-25, no 12, pp 1226-1241.

Bowie, Jack; Barnett, G. Octo (1976): "MUMPS - An Economical and Efficient Time-Sharing System for Information Management", Computer Programs in Biomedicine, vol 6, pp 11-22.

Boyce, Raymond F.; Chamberlin, Donald D. (1973): "Using a Structured English Query Language as a Data Definition Facility", IBM Research Report RJ 1318, Thomas J. Watson Research Center, Yorktown Heights, New York, December.

Bray, O.H. (1977): "Data Management Requirements: The Similarity of Memory Management, Database Systems and Message Processing", presented at the 3rd Workshop on Computer Crchitecture for Non-Numeric Processing, Syracuse U., New York, May.

Brinch Hansen, Per (1970): "The Nucleus of a Multiprogramming System", Comm. ACM, vol 13, no 4, pp 238-241,250.

Brinch Hansen (ed.), Per (1971): "RC 4000 Software Multiprogramming System", RCSL No. 55-D140, A/S Regnecentralen, Copenhagen, February.

Brinch Hansen, Per (1973a): Operating Systems Principles, Prentice-Hall, Englewood-Cliffs, New Jersey.

Brinch Hansen, Per (1973b): "Concurrent Programming Concepts", ACM Computing Surveys, vol 5, no 4, pp 223-245.

Brinch Hansen, Per (1975): "The Programming Language Concurrent Pascal", IEEE Trans. on Software Engineering, vol SE-1, no 2, pp 199-207.

Brinch Hansen, Per (1976a): "The Solo Operating System: A Concurrent Pascal Program", Software - Practice and Experience, vol 6, no 4, pp 141-149.

Brinch Hansen, Per (1976b): "The Solo Operating System: Processes, Monitors, and Classes", Software - Practice and Experience, vol 6, no 4, pp 165-200.

Burner, B.H.; Million, R.P.; Rechard, O.W.; Sobolewski, J.S. (1969): "A Programmable Data Concentrator for a Large Computer System", IEEE Trans. on Computers, vol C-18, no 11, pp 1030-1038.

Burroughs (1970a): "B6500 Information Processing Systems: Extended Algol Reference Manual", Form 1039559, Burroughs Corporation, Detroit, Michigan, January.

Burroughs (1970b): "B6500 Information Processing Systems: Espol Reference Manual", Form 1042744, Burroughs Corporation, Detroit, Michigan, January.

Burroughs (1972): "B6700 Information Processing Systems: Reference Manual", Form 1058633, Burroughs Corporation, Detroit, Michigan, May.

Buzen, Jeffrey P. (1975): "I/O Subsystem Architecture", Proc. IEEE, vol 63, no 6, pp 871-879.

Canaday, R.H.; Harrison, R.D.; Ivie, E.L.; Ryder, J.L.; Wehr, L.A. (1974): "A Back-end Computer for Data Base Management", Comm. ACM, vol 17, no 10, pp 575-582.

Casey, D.P. (1973): "Logical Data Interface", IBM Technical Disclosures Bulletin, vol 16, no 4, pp 1203-1207.

CDC (1971): "CYBER 70 Model 72 Computer Systems Reference Manual: System Description and Programming Information, vol 1", Publication No. 60347000, Control Data Corporation.

CDC (1975a): "6000 Series Computer Systems: Reference Manual", Publication No. 60100000, Control Data Corporation, February.

CDC (1975b): "7600 Series and CYBER 70 Model 76 Computer Systems: Hardware Reference Manual", Publication No. 60367200, Control Data Corporation.

CDC (1976): "Cyber 70 Series: Kronos 2.1 Workshop Reference Manual", Publication No. 97404700, Control Data Corporation, April.

Chamberlin, D.D.; Astrahan, M.M.; Eswaran, K.P.; Griffiths, P.P.; Lorie, R.A.; Mehl, J.W.; Reisner, P.; Wade, B.W. (1976): "SEQUEL 2: A Unified Approach to Data Definition, Manipulation and Control", IBM J. Research and Development, vol 20, no 6, pp 560-575.

Chen, Peter Pin-Shan (1976): "The Entity-Relationship Model - Towards a Unified View of Data", ACM Trans. on Database Systems, vol 1, no 1, pp 9-36.

Childs, D.L. (1968): "Feasibility of a Set-Theoretic Data Structure: A General Structure Based on a Reconstituted Definition of a Relation", Proc. IFIP Congress 68, Edinburgh, August, New-Holland, Amsterdam, pp 420-430.

CODASYL (1971): Data Base Task Group Report, CODASYL DBTG, April, ACM, New York.

CODASYL (1973): Data Definition Language Committee Journal of Development, CODASYL DDLC, June, IFIP Administrative Data Processing Group, Amsterdam.

Codd, E.F. (1970): "A Relational Model of Data for Large Shared Data Banks", Comm. ACM, vol 13, no 6, pp 377-387.

Computer Automation (1976): "Distributed I/O System: User's Manual", Reference 91-53629-00B1, Computer Automation, Irvine, California, August.

Cook, Thomas J. (1975): "A Data Base Management System Design Philosophy", Proc. ACM SIGMOD International Conference on the Management of Data, San Jose, California, May, ACM, New York, pp 15-22.

Cooper, Richard G. (1973): "Micromodules: Microprogrammable Building Blocks for Hardware Development", Proc. 1st Annual Symp. on Comp. Architecture, Florida, December, ACM SIGARCH Comp. Architecture News, vol 2, no 4, pp 221-226.

Copeland, George P.; Lipovski, G.J.; Su, Stanley Y.W. (1973): "The Architecture of CASSM: A Cellular System for Non-numeric Processing", Proc. 1st Annual Symp. on Comp. Architecture, Florida, December, ACM SIGARCH Comp. Architecture News, vol 2, no 4, pp 121-128.

Coulouris, G.F.; Evans, J.M.; Mitchell, R.W. (1971): "A Hardware Aided Approach to Content-addressing in Data Bases", Hardware Software Firmware Trade-offs: Proc. IEEE International Conf., Boston, September, IEEE Computer Society, California, pp 19-20.

Date, C.J.; Hopewell, P. (1971): "File Definition and Logical Data Independence", Proc. ACM SIGFIDET Workshop on Data Description, Access and Control, San Diego, California, November, ACM, New York, pp 117-138.

Date, C.J. (1976): "An Architecture for High-Level Language Database Extensions", Proc. SIGMOD International Conf. on the Management of Data, Washington, June, ACM, New York, pp 101-122.

Day, John (1973): "A Proposed File Access Protocol Specification", NIC Document # 16819, Network Information Centre, Stanford Research Institute, Menlo Park, California, June.

DEC (1970): "PDP-10 Reference Handbook", Order code: AIW, Digital Equipment Corporation, Maynard, Massachusetts.

DEC (1975): "DECSYSTEM-10: Technical Summary", Digital Equipment Corporation, Maynard, Massachusetts.

Deo, Narsingh (1974): Graph Theory with Applications to Engineering and Computing Science, Prentice-Hall Inc., Englewood Cliffs, New Jersey.

Dijkstra, Edsger W. (1968): "The Structure of the 'THE'-Multiprogramming System", Comm. ACM, vol 11, no 5, pp 341-346.

Dijkstra, E.W. (1971): "Hierarchical Ordering of Sequential Processes", Acta Informatica, vol 1, no 2, pp 115-138.

Doran, R.W. (1975): "The International Computer Ltd. ICL2900 Computer Architecture (compared to the Burroughs B6700/7700)", ACM SIGARCH Comp. Architecture News, vol 4, no 3, pp 24-47.

Downs, D.; Popek, G. (1977): "Similarities and Differences Between O/S and Data Management Security - A Study Towards a Kernel Design of Data Base Security Software", presented at IEEE Comp. Society's Workshop on Operating and Data Base Management Systems, Northwestern Univ., Illinois, March.

Endres, Albert (1975): "An Analysis of Errors and Their Causes in System Programs", IEEE Trans. on Software Engineering, vol SE-1, no 2, pp 140-149.

Eswaran, Kapali P. (1976): "Specifications, Implementations and Interactions of a Trigger Subsystem in an Integrated Database System", IBM Research Report RJ 1820, Thomas J. Watson Research Center, Yorktown Heights, New York, August.

Eswaran, K.P.; Gray, J.N.; Lorie, R.A.; Traiger, I.L. (1976): "The Notions of Consistency and Predicate Locks in a Database System", Comm. ACM, vol 9, no 11, pp 624-633.

Feiertag, R.J.; Organick, E.I. (1971): "The Multics Input/Output System", Proc. 3rd Symp. on Operating Systems Principles, Palo Alto, October, ACM SIGOPS Op. Sys. Review, 1972, vol 6, no 1&2, pp 35-41.

Fernandez, E.B.; Summers, R.C.; Lang, T.; Coleman, C.D. (1976): "Architectural Support for System Protection and Database Security", Report G320-2683, IBM Los Angeles Scientific Center, December, (IEEE Computer Society Repository R76-324).

Flores, Ivan (1969): Computer Organization, Prentice-Hall Inc., Englewood Cliffs, New Jersey.

Flynn, Michael J. (1977): "Some Remarks on High Speed Computers", Digest of Papers: 14th Spring COMPCON, San Francisco, March, IEEE Comp. Society, Long Beach, California, pp 18-20.

Gagliardi, U.O. (1975): "Trends in Computing-system Architecture", Proc. IEEE, vol 63, no 6, pp 858-862.

General Electric (1970): "GE-600 Line Integrated Data Store: Reference Manual", reference no. CPB-1565A, General Electric Corporation, April.

Gerschke, C.M.; Mitchell, J.G. (1975): "On the Problem of Uniform References to Data Structures", IEEE Trans. on Software Engineering, vol SE-1, no 2, pp 207-219.

Gray, J.N.; Lorie, R.A.; Putzolu, G.R.; Traiger, I.L. (1976): "Granularity of Locks and Degrees of Consistency in a Shared Data Base", Proc. IFIP Working Conf. on Modelling in Data Base Management Systems, G.M. Nijssen (ed.), North-Holland, Amsterdam, pp 365-394.

Hardgrave, W.T. (1975): "Set Processing in a Network Environment", ICASE Report No. 75-7, Universities Space Research Assoc., Hampton, Virginia, March, (NTIS report N75 21035).

Hawley, D.A.; Knowles, J.S.; Tozer, E. (1975): "Database Consistency and the CODASYL DBTG Proposals", Computer J., vol 18, no 3, pp 206-212.

Heacox, H.C.; Cosloy, E.S.; Cohen, J.B. (1975): "An Experiment in Dedicated Data Management", Proc. 1st International Conf. on Very Large Data Bases, Massachusetts, September, ACM, New York, pp 511-513.

Held, G.; Stonebraker, M.; Wong, E. (1975): "INGRES - A Relational Data Base Management System", Proc. AFIPS National Comp. Conf., vol 44, Anaheim, California, May, AFIPS Press, Montvale, New Jeresy, pp 409-416.

Hinke, Thomas H.; Schaefer, Marvin (1975): "Secure Data Management System", Report RADC-TR-75-266, Systems Development Corporation, Santa Monica, California, November (NTIS report AD A019 201).

Hoagland, Albert S. (1976): "Magnetic Recording Storage", IEEE Trans. on Computers, vol C-25, no 12, pp 1283-1288.

Hoare, C.A.R. (1973): "A Structured Paging System", Computer J., vol 16, no 3, pp 209-214.

Hoare, C.A.R. (1974): "Monitors : An Operating System Structuring Concept", Comm. ACM, vol 17, no 10, pp 549-557.

Honeywell (1971): "Series 200: MOD 1(MSR) Data Management Subsystem", File No. 123.6005.141C.5: Order No. 618, Honeywell Information Systems Inc., March.

Honeywell (1975): "Series 60 Level 66: Summary Description", File No. 1P01: Order No. DC64 (Rev. 1), Honeywell Information Systems Inc., Waltham, Massachusetts.

Honeywell (1976): "Multics PL/1 Language Specification", File No. 1L23: Order No. AG94 (Rev. 2), Honeywell Information Systems Inc., Waltham, Massachusetts, July.

Howie, H. Robert, Jr. (1976): "More Practical Applications of Trillion-Bit Mass Storage Systems", Digest of Papers: 12th Spring COMPCON, San Francisco, February, IEEE Comp. Society, Long Beach, California, pp 53-56.

Hsiao, David; Harary, Frank (1970): "A Formal System for Information Retrieval from Files", Comm. ACM, vol 13, no 2, pp 67-73.

Hsiao, D.K.; Kannan, K. (1977): "The Architecture of a Database Computer", presented at the 3rd Workshop on Computer Architecture for Non-Numeric Processing, Syracuse U., New York, May.

Hutt, Andrew T.F. (1974): "A Data Base Approach to System Architecture", Proc. IFIP Congress 1974, Stockholm, August, New-Holland, Amsterdam, pp 252-256.

IBM (1963): "System Operation Reference Manual: IBM 1401/1460 Data Processing Systems", form A24-3067-0, IBM Corporation.

IBM (1969): "Introduction to System/360: Direct Access Storage Devices and Organization Methods", form GC20-1649-4, IBM Corporation.

IBM (1971): "IBM System/360 Operating System: Supervisor Services", form GC28-6646, IBM Corporation, June.

IBM (1972): "IBM S/360 and S/370 ASP Version 3 Asymmetrical Multiprocessing System: General Information Manual", form GH20-1173-1, IBM Corporation.

IBM (1973a): "Reference Manual for IBM 3830 Storage Control Model 1 and IBM 3330 Disk Storage", form GA26-1592-3, IBM Corporation.

IBM (1973b): "OS/VS Data Management Services Guide", form GC26-3783-3, IBM Corporation, December.

IBM (1973c): "OS/VS Virtual Storage Access Method (VSAM): Programmer's Guide", form GC26-3838-0, IBM Corporation, December.

IBM (1974): "Information Management System, Virtual Storage (IMS/VS): General Information Manual", form GH20-1260-1, IBM Corporation, July.

IBM (1975): "Introduction to the IBM 3850 Mass Storage System (MSS)", form GA32-0028-2, IBM Corporation, July.

Jensen, E. Douglas (1975): "The Influence of Microprogramming on Computer Architecture: Distributed Processing", Proc. ACM Annual Conf., Minneapolis, Minnesota, October, ACM, New York, pp 125-128.

Jensen, E. Douglas; Thurber, Kenneth J.; Schneider, G. Michael (1976): "A Review of Systematic Methods in Distributed Processor Interconnection", Presented at IEEE International Conf. on Communications, Philadelphia, Pennsylvania, June.

Juliussen, Egil; Bhandarkar, Dileep (1975): "A Comparitive Evaluation of the Cost-Effectiveness of Computer Systems", Presented at ACM Annual Conf., Minneapolis, Minnesota, October.

Juliussen, J. Egil (1976): "Why is Peripheral Interfacing so Expensive?", Digest of Papers: 13th Fall COMPCON, Washington, September, IEEE Comp. Society, Long Beach, California, pp 274-276.

Katzman, James A. (1977): "System Architecture for Nonstop Computing", Digest of Papers: 14th Spring COMPCON, San Francisco, March, IEEE Comp. Society, Long Beach, California, pp 77-79.

King, Paul F.; Collmeyer, Arthur J. (1973): "Database Sharing - An Efficient Mechanism for Supporting Concurrent Processes", Proc. AFIPS National Comp. Conf., vol 42, New York, May, AFIPS Press, Montvale, New Jersey, pp 271-275.

Kleinrock, Leonard (1975): Queueing Systems, Volume 1, John Wiley and Sons, New York.

Lauesen, Soren (1975): "A Large Semaphore Based Operating System", Comm. ACM, vol 18, no 7, pp 377-389.

Lee, Imsong (1974): "LSI Microprocessors and Microprograms for User Oriented Machines", Supplement to the Preprints 7th Annual Workshop on Microprogramming, Palo Alto, September, ACM, New York, pp S.1-S.13

Lin, Chyuan Shiun; Smith, Dianne C.P.; Smith, John Miles (1976): "The Design of a Rotating Associative Memory for Relational Database Applications", ACM Trans. on Database Systems, vol 1, no 1, pp 53-65.

Linde, Richard H. (1975): "Operating System Penetration", Proc. AFIPS National Comp. Conf., vol 44, Anaheim, California, May, AFIPS Press, Montvale, New Jersey, pp 361-368.

Lipovski, G. Jack; Su, Stanley Y.W. (1975): "On Non-numeric Architecture", ACM SIGARCH Comp. Architecture News, vol 4, no 1, pp 14-29.

Liskov, Barbara H. (1972): "The Design of the Venus Operating System", Comm. ACM, vol 15, no 3, pp 144-149.

Liskov, Barbara; Zilles, Stephen (1974): "Programming with Abstract Data Types", ACM SIGPLAN Notices, vol 9, no 4, pp 50-59.

Lowenthal, Eugene I. (1976a): "Backend Machines for Data Base Management: A Tutorial", Proc. 5th Texas Conf. on Computing Systems, U. Texas, Austin, October, IEEE Comp. Society, Long Beach, California, pp 21-25.

Lowenthal, Eugene I. (1976b): "The Backend (Data Base) Computer - Parts I and II", Auerbach Data Base Management Series, 24-10-04 and 24-01-05, Auerbach Publishers, New Jersey.

Lowenthal, Eugene I. (1977): "A General Purpose DBMS Kernel", presented at the 1977 USAFA Computer Related Information Systems Symposium (CRISYS), Colorado Springs, Colorado, January.

Macri, Philip P. (1976): "Deadlock Detection and Resolution in a CODASYL Based Data Management System", Proc. ACM SIGMOD International Conf. on Management of Data, Washington, June, ACM, New York, pp 45-49.

Madnick, Stuart E.; Alsop, Joseph W. (1969): "A Modular Approach to File System Design", Proc. AFIPS Spring Joint Comp. Conf, vol 34, Boston, Massachusetts, May, AFIPS Press, Montvale, New Jersey, pp 1-13.

Madnick, Stuart E.; Donovan, John J. (1973): "Application and Analysis of the Virtual Machine Approach to Information System Security and Isolation", Proc. SIGOPS-SIGARCH Workshop on Virtual Computer Systems, Harvard U., March, ACM, New York, pp 210-224.

Madnick, Stuart E.; Donovan, John J. (1974): Operating Systems, McGraw-Hill Book Co., New York.

Madnick, Stuart E. (1975): "Design of a General Hierarchical Storage System", presented at IEEE International Convention and Exposition (INTERCON), April, New York.

Manola, Frank A. (1975): "Principles of the CODASYL Approach to the Description of Data Structures", NRL Memorandum Report 3068, Naval Research Laboratory, Washington D.C., June.

Manola, F.A. (1976): "The GODASYL Data Description Language: Status and Activities, April 1976", NRL Memorandum Report 8038, Naval Research Laboratory, Washington D.C., November, (NTIS report AD A033 401).

Marill, Thomas; Stern, Dale (1975): "The Datacomputer - A Network Data Utility", Proc. AFIPS National Comp. Conf., vol 44, Anaheim, California, May, AFIPS Press, Montvale, New Jersey, pp 389-395.

Martin, R.R.; Frankel, H.D. (1975): "Electronic Disks in the 1980's", Computer, vol 8, no 2, pp 24-30.

Maryanski, Fred J. (1977): "Performance of Multi-Processor Back-End Data Base Management Systems", Technical Report CS 77-07, Dept. of Computer Science, Kansas State University, Manhattan, Kansas, April.

Maryanski, F.; Fisher, P.; Wallentine, V. (1975): "Usability and Feasibility of Back-End Minicomputers", Report under USACSC Grant No. DAHC04-75-G-0137, U.S. Army Computer Systems Command, Fort Belvoir, Virginia, June.

Maryanski, Fred J.; Fisher, Paul S.; Wallentine, Virgil E. (1976): "Evaluation of Conversion to a Back-End Data Base Management System", Proc. ACM Annual Conf., Houston, Texas, October, ACM, New York.

Maryanski, Fred J.; Fisher, Paul S.; Wallentine, Virgil E.; Calhoun, Myron A.; Sernovitz, Louis (1976): "A Minicomputer Based Distributed Data Base System", Proc. Trends and Applications Symp.: Micro & Mini Systems, Gaithersburg, Maryland, May, IEEE Comp. Soc., Long Beach, California.

Maryanski, Fred J.; Fisher, Paul S.; Wallentine, Virgil E. (1976): "A User-Transparent Mechanism for the Distribution of a CODASYL Data Base Management System", Technical Report CS 76-22, Dept. of Computer Science, Kansas State University, Manhattan, Kansas, December.

Maryanski, Fred J.; Wallentine, Virgil E. (1976): "A Simulation Model of a Back-End Data Base Management System", Proc. of the 7th Pittsburgh Conference on Modeling and Simulation, April.

McDonell, K.J.; Marsland, T.A. (1977): "The Michian Terminal System: Internal Architecture", Technical Report TR77-3, Dept. of Computing Science, U. of Alberta, September.

McGregor, D.R.; Thomson, R.G.; Dawson, W.N. (1976): "High Performance Hardware for Database Systems", Proc. 2nd International Conf. on Very Large Data Bases, Brussels, Belgium, September, ACM, New York.

Melliar-Smith, P.M.; Randell, B. (1977): "Software Reliability: The Role of Programmed Exception Handling", Proc. ACM Conf. on Language Design for Reliable Software, N. Carolina, March, (SIGOPS Op. Sys. Review, vol 11, no 2), pp 95-100.

Minsky, Naftaly (1974): "Another Look at Data Bases", ACM SIGMOD FDT Bulletin, vol 6, no 4, pp 9-17.

Miller, Stephen W.; Gagliardi, Ugo O. (1976): Final Report from the Symposium on Advanced Memory Concepts, Symposium held at Stanford Research Institute, June, (NTIS Report AD A029 631).

Mitchell, R.W. (1976): "Content Addressable File Store", presented at Online Conf. on Database Technology, London, England, April.

Moreira, Alberto Cezar de Souza; Pinheiro, Claudio; D'Elia, Luiz Fernand (1974): "Integrating Database Management into Operating Systems - An Access Method Approach", Proc. AFIPS National Comp. Conf., vol 43, Chicago, Illinois, May, AFIPS Press, Montvale, New Jersey, pp 57-62.

Nijssen, G.M. (1972): "Common Data Base Languages", ACM SIGBDP Data Base, vol 4, no 4, pp 7-11.

Nijssen, G.M. (1975): "Two Major Flaws in the CODASYL DDL 1973 and Proposed Corrections", Information Systems, vol 1, no 4, pp 115-132.

Ohmori, K.; Koike, N.; Nezu, K.; Suzuki, S. (1974): "MICS - A Multi-microprocessor System", Proc. IFIP Congress 1974, Stockholm, August, North-Holland, Amsterdam, pp 98-102.

Olle, T. William (1974): "Current and Future Trends in Data Base Management Systems", Proc. IFIP Congress 1974, Stockholm, August, North-Holland, Amsterdam, pp 998-1006.

Olle, William T. (1975): "A Practitioner's View of Relational Data Base Theory", ACM SIGMOD FDT Bulletin, vol 7, no 3&4, pp 29-43.

Omahen, Kenneth (1975): "Estimating the Response Time for Auxiliary Memory Configurations with Multiple Movable-Head Disk Modules", Proc. 1st International Conf. on Very Large Data Bases, Masachusetts, September, ACM, New York, pp 473-495.

Organick, Elliot I. (1972): The Multics System: An Examination of Its Structure, MIT Press, Cambridge, Massachusetts.

Ozkarahan, E.A.; Schuster, S.A.; Smith, K.C. (1975): "RAP - An Associative Processor for Data Base Management", Proc. AFIPS National Comp. Conf., vol 44, Anaheim, California, May, AFIPS Press, Montvale, New Jersey, pp 379-387.

Palmer, Ian R. (1974): "Levels of Database Description", Proc. IFIP Congress 1974, Stockholm, August, North-Holland, Amsterdam, pp 1031-1036.

Papadimitriou, Christos H.; Bernstein, Philip A.; Rothnie, James B. (1977): "Some Computational Problems Related to Database Concurrency Control", an unpublished manuscript.

Parnas, David (1974): "On a 'Buzzword': Hierarchical Structure", Proc. IFIP Congress 1974, Stockholm, August, New-Holland, Amsterdam, pp 336-339.

Patel, Rajini M. (1969): "Basic I/O Handling on Burroughs B6500", Proc. 2nd Symposium on Operating Systems Principles, Princeton, New Jersey, October, ACM, New York, pp 120-129.

Pirkola, Gary C. (1975): "A File System for a General-Purpose Time-Sharing Environment", Proc. IEEE, vol 63, no 6, pp 918-924.

Popek, G.J.; Kline, C.S. (1974): "Verifiable Secure Operating System Software", Proc. AFIPS National Comp. Conf., vol 43, Chicago, May, AFIPS Press, Montvale, New Jersey, pp 145-151.

Poujoulat, G.H. (1974): "Microprogramming of a Burst Structure", Preprints 7th Annual Workshop on Microprogramming, Palo Alto, California, September, ACM, New York, pp 48-51.

Randell, B. (1975): "System Structure for Software Fault Tolerance", Proc. International Conf. on Reliable Software, Los Angeles, April, ACM SIGPLAN Notices, vol 10, no 6, pp 437-449.

Reiter, Allen (1975): "Data Models for Secondary Storage Representations", MRC Technical Summary Report #1554, Mathematics Research Centre, U. of Wisconsin, Madison, July, (NTIS report AD A016 347).

Rice, Rex (1970): "LSI and Computer Systems Architecture", Computer Design, vol 9, December, pp 57-63.

Ritchie, Dennis M. (1973): "C Reference Manual", Bell Telephone Laboratories, Murray Hill, New Jersey.

Ritchie, Dennis M.; Thompson, Ken (1974): "The UNIX Time-Sharing System", Comm. ACM vol 17, no 7, pp 365-375.

Robinson, K.A. (1975): "DMS 1100: An Indepth Evaluation", Software World, vol 6, no 2, pp 8-14.

Rodriguez-Rosell, J.; Eckhouse, R. (1977): "Management of Data by Future Operating Systems", New Directions for Operating Systems: A Workshop Report, J.C. Browne (ed.), ACM SIGOPS Op. Sys. Review, vol 11, no 1, pp 23-25.

Rosenthal, Robert S. (1977): "An Evaluation of Backend Data Base Management Machines", presented at the 1977 USAFA Computer Related Information Systems Symposium (CRISYS), Colorado Springs, Colorado, January.

Rosenthal, Robert S. (1977b): private communication, MRI Systems Corporation, Austin Texas, May.

Schiller, W.L. (1975): "Design of a Security Kernel for the PDP-11/45", Report ESD-TR-75-69, MITRE Corp., Bedford, Massachusetts, May, (NTIS report AD A011 712).

Schlageter, G. (1976): "The Problem of Lock By Value in Large Data Bases", Computer J., vol 19, no 1, pp 17-20.

Schlageter, Gunter (1975): "Access Synchronization and Deadlock Analysis in Database Systems: An Implementation Oriented Approach", Information Systems, vol 1, pp 97-102.

Schroeder, Michael D. (1975): "Engineering a Security Kernel for Multics", Proc. 5th Symp. on Operating Systems Principles, U. Texas, Austin, ACM SIGOPS Operating Systems Review, vol 9, no 5, pp 25-32.

Senko, M.E.; Altman, E.B.; Astrahan, M.M.; Fehder, P.L. (1973): "Data Structures and Accessing in Data-base Systems", IBM Systems J., vol 12, no 1, pp 30-93.

Shemer, J.E.; Collmeyer, A.J. (1972): "Database Sharing: A Study of Interference, Roadblock and Deadlock", Proc. ACM SIGFIDET Workshop on Data Description, Access and Control, Denver, Colorado, November, ACM, New York, pp 147-164.

Sibley, Edgar H. (1974): "On the Equivalences of Data Based Systems", Proc. ACM SIGMOD Workshop on Data Description, Access and Control, vol 2, Ann Arbour, Michigan, May, ACM, New York, pp 45-76.

Sibley, E.H. (ed.) (1976): "Data-Base Management Systems", ACM Computing Surveys, vol 8, no 1, (Special Issue).

Sindelar, Frank; Hoffman, Lance J. (1974): "A Two Level Disk Protection System", Memo ERL-M452, College of Engineering, U. of California, Berkley, May, (IEEE Comp. Society Repository R75-86).

Snuggs, Mary E.; Popek, Gerald J.; Peterson, Ronald J. (1975): "Data Base System Objectives as Design Constraints", Proc. ACM Annual Conf., Minneapolis, Minnesota, October, ACM, New York, pp 641-647.

STIS (1976): private communication, Set Theoretic Information Systems Corporation, Ann Arbor, Michigan, September.

Stonebraker, Michael; Held, Gerald (1975): "Networks, Hierarchies, and Relations in Data Base Management Systems", presented at the 1975 ACM Pacific Conf., (Memorandum No. ERL-M504, Electronics Research Lab, U. California, Berkeley, March).

Summers, Rita C.; Coleman, Charles D.; Fernandez, Eduardo B. (1974): "A Programming Language Approach to Secure Data Base Access", Technical report G320-2662, IBM Los Angeles Scientific Center, California, May.

Tao, W.Y.Y. (1974): "A Firmware Data Compression Unit", Report UIUCDCS-R-74-617, Dept. Computer Science, U. of Illinois, Urbana, January.

Tomlin, E.L. (1973): "Microprogrammed Disc Controllers", M.Sc. Thesis, Naval Postgraduate School, Monterey, California, December, (NTIS report AD 789 812).

Tsichritzis, D. (1976): "LSL: A Link Selector Language", Proc. SIGMOD International Conf. on the Management of Data, Washington, June, ACM, New York, pp 123-133.

University of Michigan (1973): "The Michigan Terminal System, Volume 3: Subroutine and Macro Descriptions", The University of Michigan Computing Center, Ann Arbor, Michigan, May.

Wallentine, V.; Maryanski, F.; Fisher, P.; McBride, R.; Fox, S.; Chapin, W.; Allen, L. (1975): "Technical Report on the Implementation of a Backend Data Base Management System", Technical Report TR-CS-09-75, Dept. of Computer Science, Kansas State University, Manhattan, Kansas, October.

Wasserman, Anthony Ira (1976): "Embedding Data Management Operations in Programming Languages", Digest of Papers: 12th Spring COMPCON, San Francisco, February, IEEE Comp. Society, Long Beach, California, pp 79-82.

Withington, Frederick G. (1975): "Beyond 1984: A Technology Forecast", Datamation, vol 21, no 1, pp 54-73.

Withington, Frederic G. (1976): "Future Computer Technology", ACM SIGBDP Newsletter: Data Base, vol 7, no 4, pp 7-14.

Winter, Richard; Hill, Jeffrey; Greiff, Warren (1973): "Further Datalanguage Design Concepts", Computer Corporation of America, Cambridge, Massachusetts, December.

White, J.C.C. (1975): "Design of a Secure File Management System", Report ESD-TR-75-57, MITRE Corp., Bedford, Massachusetts, April, (NTIS report AD A010 590).

White, Lionel S.; Welch, T.A. (1975): "Analysis of Virtual Memory Implementation", Technical report TR 174, Information Systems Research Laboratory, Univ. of Texas, July, (NTIS report AD A023 116).

Whitney, Kevin M. (1973): "Fourth Generation Data Management Systems", Proc. AFIPS National Comp. Conf., vol 42, New York, May, AFIPS Press, Montvale, New Jersey, pp 239-244.

Wulf, W.A.; Russell, D.; Habermann, A.N.; Geschke, C.; Apperson, J.; Wile, D.; Brender, R. (1971): "BLISS Reference Manual", Computer Science Department, Carnegie-Mellon Univ., Pittsburgh, Pennsylvania, October.

Wulf, W.; Cohen, E.; Corwin, W.; Jones, A.; Levin, R.; Pierson, C.; Pollack, F. (1974): "HYDRA: The Kernel of a Multiprocessor Operating System", Comm. ACM, vol 17, no 6, pp 337-345.

APPENDIX A

A PROPOSAL FOR THE IMPLEMENTATION OF A SPOOLING SUBSYSTEM USING A COMPLEX INPUT-OUTPUT INTERFACE

The example presented in this Appendix is intended to illustrate a possible level of input-output interaction across a complex logical interface. The skeletal design of a hypothetical spooling subsystem will be presented; a spooling subsystem was chosen since this is one application which has historically been implemented using a low level interface and a tight coupling between the actual devices and the spooling routines.

The particular interface which has been selected is the proposed uniform interface described in Section 4.3, which is in turn based upon the CODASYL Data Definition (DDL) and Data Manipulation Languages (DML). Interface schema definitions (Figures A.1, A.2, A.3 and A.4) based on the modified CODASYL DDL will be used to describe the data structures accessed and manipulated by user processes and the spooling routines during spooling operations.

It is not intended that this proposal be critically evaluated with respect to its run-time efficiency, rather the aim is to show that a logically consistent view of input-output operations assists, rather than hinders the sound development of cooperating, concurrent software modules. If the desirability and feasibility of the concept can been established, then attention may be turned to details of efficiency and implementation (refer to Chapters 5 and 6).

A.1 Data Structures and Declarations

The hypothetical spooling subsystem (described in the following sections) requires access to three logical partitions of the total secondary storage resources, each of which is described by a corresponding interfaces schema:

(1) USER-CONTROL: this partition holds the user identification and description data for valid users, accounting information, user validation passwords, etc. Only the unique user identification field (USER-ID) of the USER-NAME record type is visible to the spooling subsystem (see Figure A.1), since it will be assumed that the input-output accounting and user validation functions are performed elsewhere.

(2) FILE-SYSTEM: the directories, access information, file descriptions and files for the user accessible secondary storage data structures are held in the partition described by this schema (see Figure A.2). The spooling subsystem has access to two permanent set types in this area, namely USER-DIRECTORY which associates zero, one or more FILE-ID's with a USER-ID, and WORK-FILE, where

each occurrence is a set of TEXT records which are uniquely identifier by their line numbers -- e.g. a 'line' file under the Michigan Terminal System. All the pseudo device files are stored in this region.

(3) DEVICE-CONTROL: the data held in this region is dependent upon the hardware device configuration, and it is used by the spooling subsystem to identify groups of identical devices (i.e. occurrences of the DEVICES set type), the physical device attributes and the individual devices (see Figure A.3). Note, the devices described in this area are the <u>spooled</u> devices, <u>not</u> the secondary storage devices. In addition, no process outside the spooling subsystem may use this schema, or access one of the spooled devices directly.

```
SCHEMA name is USER-CONTROL
  ;CALL ... ON ERROR during ...
  ;ACCESS-CONTROL lock for ... is ...

RECORD name is USER-NAME
  ;IDENTIFIER is USER-ID
  ;CALL ... ON ERROR during . .
  ;ACCESS-CONTROL lock for ... is ...
    01  USER-ID                  TYPE is CHARACTER ...
```

Figure A.1 Interface Schema Declarations for the User Identification Data

In addition, the spooling subsystem maintains three temporary set types, linking together records in the three previously mentioned areas (see Figure A.4). These sets are used to implement a queue of outstanding requests for files to be spooled to output devices (SPOOL-QUEUE), to associate user information with spooled files (SPOOL-DIRECTORY), and to record the allocation of a specific spooled file to a particular device (DEVICE-ASSIGNMENT).

Using these interface schema declarations, it is possible to specify the derived subschemata appropriate for an input spooling process, an output spooling process and a user process performing spooled input-output. Data structure diagrams will be used to describe those sections of the database which a particular subschema makes visible to its associated process. Where necessary, the procedural operation of a process will be illustrated using the modified CODASYL DML[1].

A.2 An Input Spooling Process

The input spooling process's view of the database is illustrated in Figure A.5, and the associated procedure for a typical input spooling process (in this case, servicing a card reader) is outlined in Figure A.6.

1: While COBOL is not the ideal language for this exercise, at the present time, the operations outlined in Section 4.3.2 constitute a complete set of DML capabilities - undoubtedly, any systems programming language designed to interface with the proposed input-output module would support a set of DML primitives which would be, at least, semantically equivalent to the proposed DML facilities.

```
SCHEMA name is FILE-SYSTEM
  ;CALL ... ON ERROR during ...
  ;ACCESS-CONTROL lock for ... is ...

RECORD name is FILE-NAME
  ;IDENTIFIER is OWNER-ID,FILE-ID
  ;CALL ... ON ERROR during ...
  ;ACCESS-CONTROL lock for ... is ...
    01  OWNER-ID                TYPE is CHARACTER ...
    01  FILE-ID                 TYPE is CHARACTER ...
    01  FILE-ATTRIBUTES         TYPE is CHARACTER ...

RECORD name is TEXT
  ;IDENTIFIER is LINE-NUMBER WITHIN WORK-FILE set
  ;CALL ... ON ERROR during ...
  ;ACCESS-CONTROL lock for ... is ...
    01  LINE-NUMBER             TYPE is FLOAT DECIMAL ...
    01  LINE-DATA               TYPE is CHARACTER ...

SET name is WORK-FILE
  ;OWNER is FILE-NAME
  ;CALL ... ON ERROR during ...
  ;ACCESS-CONTROL lock for ... is ...
MEMBER is TEXT FIXED AUTOMATIC
  ;SET SELECTION is thru CURRENT of OWNER
  ;CALL ... ON ERROR during ...
  ;ACCESS-CONTROL lock for ... is ...

SET name is USER-DIRECTORY
  ;OWNER is USER-NAME
  ;CALL ... ON ERROR during ...
  ;ACCESS-CONTROL lock for ... is ...
MEMBER is FILE-NAME OPTIONAL MANUAL
  ;SET SELECTION is thru USER-ID in OWNER EQUAL to
   OWNER-ID of MEMBER
  ;CALL ... ON ERROR during ...
  ;ACCESS-CONTROL lock for ... is ...
```

Figure A.2 Interface Schema Declarations for the General Purpose Files and User File Directories

```
SCHEMA name is DEVICE-CONTROL
   ;CALL ... ON ERROR during ...
   ;ACCESS-CONTROL lock for ... is ...

RECORD name is DEVICE-CLASS
   ;IDENTIFIER is DEVICE-TYPE
   ;CALL ... ON ERROR during ...
   ;ACCESS-CONTROL lock for ... is ...
     01  DEVICE-TYPE               TYPE is CHARACTER ...
     01  DEVICE-ATTRIBUTES         TYPE is CHARACTER ...

RECORD name is DEVICE-NAME
   ;IDENTIFIER is HARDWARE-DEVICE-ADDRESS
   ;CALL ... ON ERROR during ...
   ;ACCESS-CONTROL lock for ... is ...
     01  DEVICE-ID                 TYPE is CHARACTER ...
     01  HARDWARE-DEVICE-ADDRESS   TYPE is CHARACTER ...

SET name is DEVICES
   ;OWNER is DEVIC  CLASS
   ;CALL ... ON ERROR during ...
   ;ACCESS-CONTROL lock for ... is ...
MEMBER is DEVICE-NAME MANDATORY AUTOMATIC linked to OWNER
   ;SET SELECTION is thru CURRENT of OWNER
   ;CALL ... ON ERROR during ...
   ;ACCESS-CONTROL lock for ... is ...

SET name is DEVICE-ASSIGNMENT
   ;OWNER is DEVICE-NAME
   ;CALL ... ON ERROR during ...
   ;ACCESS-CONTROL lock for ... is ...
MEMBER is FILE-NAME OPTIONAL MANUAL linked to OWNER
   ;SET SELECTION is thru CURRENT of OWNER
   ;CALL ... ON ERROR during ...
   ;ACCESS-CONTROL lock for ... is ...
```

Figure A.3 Interface Schema Declarations for the Device Configuration Data

```
SCHEMA name is SPOOLER
  ;CALL ... ON ERROR during ...
  ;ACCESS-CONTROL lock for ... is ...

SET name is SPOOL-DIRECTORY
  ;OWNER is USER-NAME
  ;CALL ... ON ERROR during ...
  ;ACCESS-CONTROL lock for ... is ...
MEMBER is FILE-NAME OPTIONAL MANUAL
  ;SET SELECTION is thru USER-ID in OWNER EQUAL to
   OWNER-ID of MEMBER
  ;CALL ... ON ERROR during ...
  ;ACCESS-CONTROL lock for ... is ...

SET name is SPOOL-QUEUE
  ;OWNER is DEVICE-CLASS
  ;CALL ... ON ERROR during ...
  ;ACCESS-CONTROL lock for ... is ...
MEMBER is FILE-NAME OPTIONAL MANUAL
  ;SET SELECTION is thru CURRENT of OWNER
  ;CALL ... ON ERROR during ...
   ACCESS-CONTROL lock for ... is ...
```

Figure A.4 Interface Schema Declarations for the Temporary Sets Maintained by the Spooling Subsystem

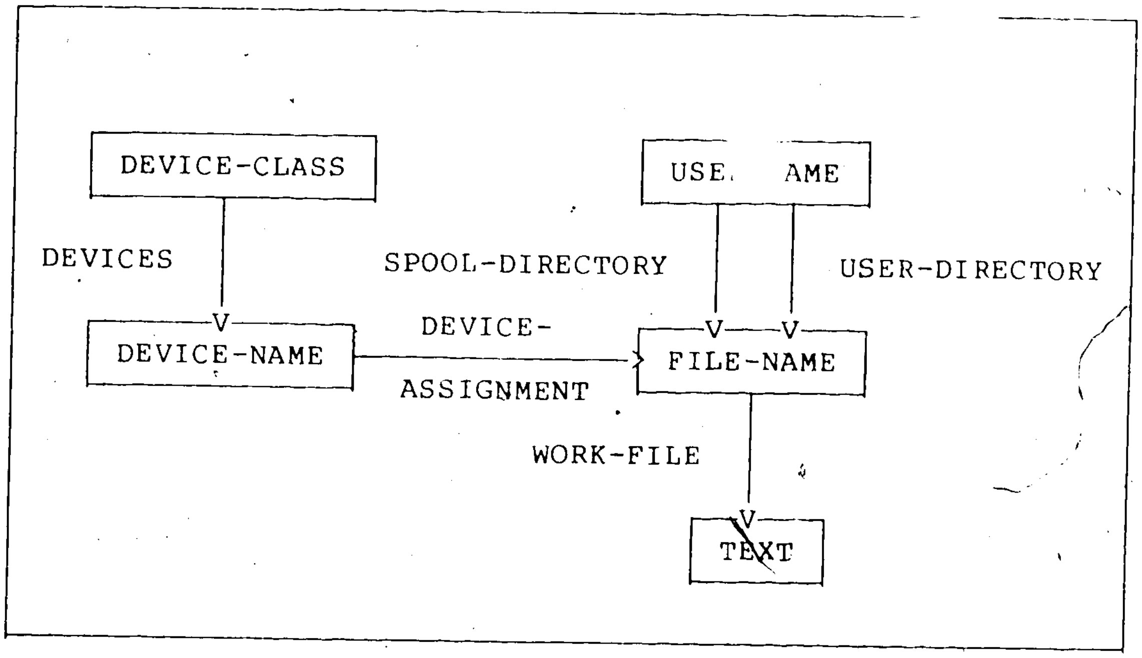

Figure A.5 Subschema for an Input Spooling Process

For the sake of simplicity, it will be assumed that each spooled device is 'driven' by a separate spooling routine -- in reality, re-entrant code, or one spooling 'monitor' for each class of spooled devices would be used. But, in this simplified example, each spooling process is concerned with only a single occurrence of the DEVICE-NAME record type (corresponding to the single device being serviced by the process).

On input, a spooled device, or indeed an individual spooled file, maybe associated with either a batch job or a non-batch job. Non-batch jobs are data files required as input to a process other than the system command language processor - they 'belong' to the process which requires the data file as input and should be stripped of all 'start-of-job' and 'end-of-job' system commands during spooling. On the other hand, batch jobs 'belong' to the system command language processor, and all system commands must be included in the spooled file. This presents no real problem (assuming a sensible system command language in which the commands may be readily identified), and the operation of both types of input spooling processes will be discussed.

The sequential execution of the input spooling process shown in Figure A.6 may be summarized as follows,

1. Identify the next USER-ID: normally the use 's unique identification is given in the first input record.
2. Determine the eventual owner of the spooled file:
   non-batch: the USER-ID, from step 1.
   batch: the special USER-ID associated with the system command language processor (e.g. *BATCH*).
3. Check that the owner exists: find USER-ID record.
4. Extract the FILE-ID: this name must be unique within USER-ID, so that the FILE-NAME can eventually be added to the approriate USER-DIRECTORY.
   non-batch: the FILE-NAME is either explicitly given (with optional FILE-ATTRIBUTES) via a system command, or the FILE-NAME and FILE-ATTRIBUTES may be assigned default values, based upon the DEVICE-TYPE.
   batch: the FILE-NAME is artificially constructed to be unique for each batch job submitted (e.g. a serially assigned receipt number).
5. Assign the device to the file: insert the FILE-NAME into the DEVICE-ASSIGNMENT set for this device - this set may

contain at most one FILE-NAME record.

6. Establish the owner,file-name association: insert the FILE-NAME into the SPOOL-DIRECTORY for the owner's USER-ID, and lock the file.
7. Do it: input records serially from the spooled device and store them in the WORK-FILE set for the current FILE-NAME. This operation ends when an 'end-of-job' system command is encountered in the input stream.
8. Make file available to owner: remove FILE-NAME from the SPOOL-DIRECTORY set (the spooling process has finished with it) and the DEVICE-ASSIGNMENT set (the device is 'free'), then insert the FILE-NAME into the owner's USER-DIRECTORY, and unlock the file. At this point, any process executing under the owner's USER-ID may access the spooled file.

A.3 An Output Spooling Process

As shown in the previous section, an input spooling process may handle at most one spooled file per spooled device (since the devices are sequential by nature). However, the number of active files under the control of an output spooling process may exceed the num[illegible] of spooled devices serviced by the process. As a res[illegible], the output spooling process is a little more complex, and in the f lowing example it has been split into two concurrent bp[illegible]cesses, namely a (device independent) Dispatcher and an [illegible]put Spooler (in this case, for a single line printer).

```
INITIALIZE.
    NOTE open the necessary schemata.
    OPEN DEVICE-CONTROL, USER-CONTROL, FILE-SYSTEM,
    SPOOLER.
    NOTE this routine is configured to drive the card
         reader at hardware address "CR01".
    FIND DEVICE-CLASS RECORD USING
       DEVICE-TYPE = "CARD-READER".
    GET DEVICE-CLASS.
    FIND DEVICE-NAME RECORD USING
       HARDWARE-DEVICE-ADDRESS = "CR01".
    GET DEVICE-NAME.
    NOTE at this point, we have the correct record
         occurrence as CURRENT of DEVICE-CLASS and
         DEVICE-NAME.
NEW-JOB.
    NOTE fetch next input record, identify new user and
         set up owner's id in TEMP-OWNER.
                 . . . . . . . . . . . . .
    FIND USER-NAME RECORD USING USER-ID = TEMP-OWNER.
    NOTE determine the FILE-ID and construct appropriate
         FILE-ATTRIBUTES.
                 . . . . . . . . . . . . .
    STORE FILE-NAME.
    LOCK FILE-NAME, WORK-FILE USAGE is EXCLUSIVE UPDATE.
    INSERT FILE-NAME INTO DEVICE-ASSIGNMENT,
    SPOOL-DIRECTORY.
SPOOL.
    NOTE this is where the spooled file is generated;
         fetch input card images one at a time (into
         LINE-DATA), construct LINE-NUMBER; branch to
         SPOOL-EOF at logical "end-of-file" on input.
                 . . . . . . . . . . . . .
    STORE TEXT; GO TO SPOOL.
SPOOL-EOF.
    NOTE make the file part of the regular file system,
         accessible from the USER-DIRECTORY set.
    FIND OWNER record of WORK-FILE set..
    REMOVE FILE-NAME FROM
    SPOOL-DIRECTORY,DEVICE-ASSIGNMENT;
    INSERT FILE-NAME INTO USER-DIRECTORY.
    UNLOCK FILE-NAME, WORK-FILE.
    NOTE go back and start all over again for next job.
    GO TO NEW-JOB.
```

Figure A.6 An Input Spooling Procedure

A.3.1 The Dispatcher

Since the Dispatcher handles the scheduling of spooled files for all devices, when called from a user process it must be passed three parameters (DEVICE-TYPE, USER-ID and FILE-ID) to specify 'which file is to be spooled where'.

As the following sequence shows, the Dispatcher is a relatively simple process (refer also to Figures A.7 and A.8),

1. Remove file from owner's control: remove FILE-NAME from the owner's USER-DIRECTORY. Once this has been done the spooled file is no longer accessible to processes executing under the owner's USER-ID - in fact the file cannot be accessed from <u>any</u> USER-DIRECTORY!
2. Schedule the file for spooling: insert the FILE-NAME into the SPOOL-QUEUE set for the cited DEVICE-TYPE. The order of the member records in this set determines the sequence in which files will be output - in this example the scheduling discipline is FCFS.
3. Preserve the owner-file name association: this association was maintained in the USER-DIRECTORY and it is preserved by inserting the FILE-NAME into the file SPOOL-DIRECTORY; the SPOOL-DIRECTORY set type is not visible to any process outside the spooling subsystem.

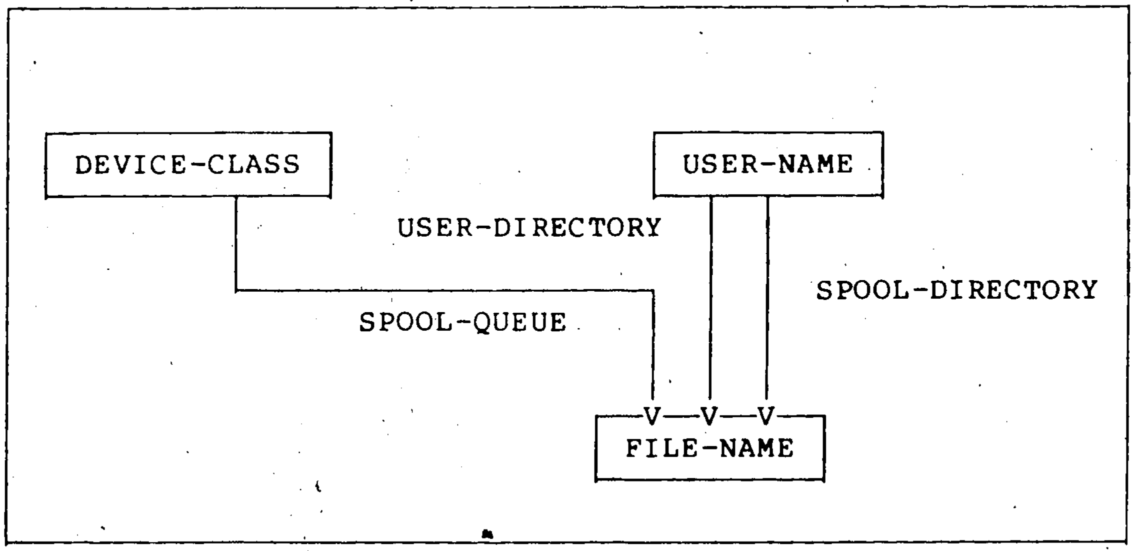

Figure A.7 Subschema for the Dispatcher

```
NOTE a procedure entry with 3 parameters;
     the desired schemata are assumed already OPENed
     by this process;
     the parameter values are stored in local
     temporary areas.
     . . . . . . . . . . . . .
FIND DEVICE-CLASS RECORD USING
   DEVICE-TYPE = parameter-1.
FIND USER-NAME RECORD USING USER-ID = parameter-2.
FIND FILE-NAME USING OWNER-ID = parameter-2,
   FILE-ID = parameter-3.
LOCK FILE-NAME USAGE is EXCLUSIVE UPDATE.
NOTE make the file inaccessible from the
     USER-DIRECTORY, then include it in the
     SPOOL-DIRECTORY and the SPOOL-QUEUE sets.
REMOVE FILE-NAME FROM USER-DIRECTORY;
INSERT FILE-NAME INTO SPOOL-DIRECTORY;
UNLOCK FILE-NAME.
INSERT FILE-NAME INTO SPOOL-QUEUE;
NOTE finished.
```

Figure A.8 A Procedure for Dispatching Output Spooled Files

A.3.2 The Output Spooler

The operation of the Output Spooler is very similar to the input spooling process described previously in Section A.2, however there are two significant differences,

(1) The output process is 'idle' whenever the SPOOL-QUEUE set associated with the particular DEVICE-CLASS is empty (i.e. when no files are awaiting output on this DEVICE-TYPE). Consequently, it will be assumed that suitable synchronization primitives are available to ensure the correct execution of this 'wait' operation. Once the spooling process has something to do, it dequeues the file, assigns the file to the device and uses the SPOOL-DIRECTORY set to identify the owner.

(2) Once all the TEXT records in the WORK-FILE have been output, the spooled file is available to be <u>deleted from the database</u>. This is necessary to avoid 'clogging up' the database with old spooled output files which are inaccessible from user processes.

A database subschema and the necessary DML commands for the hypothetical Output Spooler are shown in Figures A.9 and A.10 respectively.

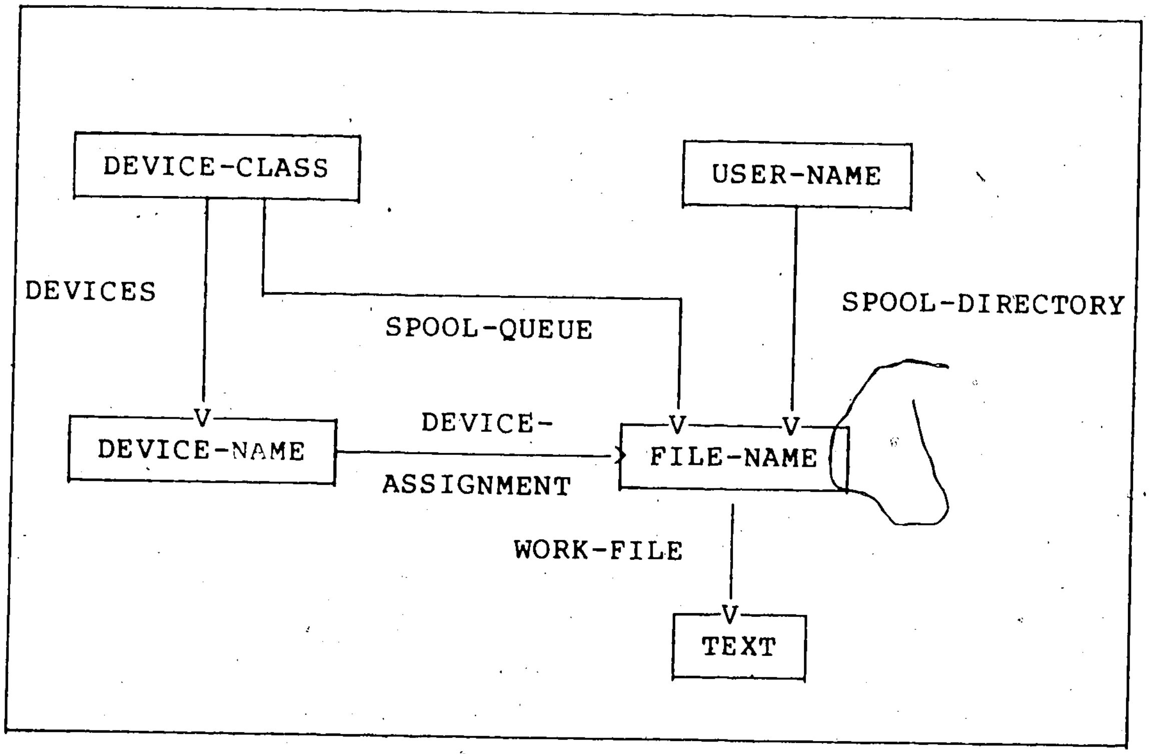

Figure A.9 Subschema for the Output Spooler

A.4 Interaction with the Spooled Device

Upto this point, no mention has been made of 'how' a spooling process achieves the transfer of information between the database management system record buffers in the User Work Area and the physical (spooled) device. A possible approach is to make the necessary low level input-output operations part of the spooling process, and thus allow direct interaction with the device.

Assuming all devices within a particular class are either spooled or non-spooled, then this technique results in a rigid hierarchic relationship between a spooling procedure and its associated low level routines. For

```
INITIALIZE.
    NOTE open the necessary schemata.
    OPEN DEVICE-CONTROL, USER-CONTROL, FILE-SYSTEM,
    SPOOLER.
    NOTE this routine is configured to drive the line
         printer at hardware address "PR01".
    FIND DEVICE-CLASS RECORD USING
       DEVICE-TYPE = "LINE-PRINTER".
    GET DEVICE-CLASS.
    FIND DEVICE-NAME RECORD USING
       HARDWARE-DEVICE-ADDRESS = "PR01".
    GET DEVICE-NAME.
    NOTE at this point, we have the correct record
         occurrence as CURRENT of DEVICE-CLASS and
         DEVICE-NAME.
WAIT-FOR-SOMETHING-TO-DO.
    IF SPOOL-QUEUE SET EMPTY THEN
    NOTE suspend execution of this process, waiting for a
         file to be spooled to a line-printer.
                    . . . . . . . . . . . .
       FIND OWNER OF SPOOL-QUEUE SET;
    ELSE NEXT SENTENCE.
    FIND NEXT FILE-NAME RECORD of SPOOL-QUEUE SET.
    LOCK FILE-NAME USAGE is EXCLUSIVE UPDATE.
    NOTE we're ready to go; remove the FILE-NAME from the
         SPOOL-QUEUE set, and place it in the
         DEVICE-ASSIGNMENT set.
    REMOVE FILE-NAME FROM SPOOL-QUEUE SET;
    INSERT FILE-NAME INTO DEVICE-ASSIGNMENT SET.
    UNLOCK FILE-NAME.
    NOTE determine the USER-ID, and output identifying
         information.
    FIND OWNER SPOOL-DIRECTORY SET.
    GET USER-NAME.
                    . . . . . . . . . . . . . .
OUTPUT-LOOP.
    FIND NEXT TEXT RECORD of WORK-FILE SET; IF
    ERROR-STATUS  QUALS END-OF-SET, GO TO OUTPUT-END.
    GET TEXT.
    NOTE output this line.
                    . . . . . . . . . . . . . .
    GO TO OUTPUT-LOOP.
OUTPUT-END.
    NOTE tidy up by deleting the FILE-NAME record, and
         (by implication) all the TEXT records.
    FIND OWNER record of WORK-FILE set.
    DELETE FILE-NAME.
    GO TO WAIT-FOR-SOMETHING-TO-DO.
```

Figure A.10 An Output Spooling Procedure

example, the spooling process for card readers is the only process in the system which may call routines to control and service the hardware interface to the card reader - all other processes execute card input operations on a pseudo card reader, via the input-output module.

A.5 User Processes

Each user has access to a single USER-DIRECTORY set, defined in the FILE-SYSTEM schema. A spooled file is stored as a single occurrence of the WORK-FILE set type, with a unique FILE-ID in the FILE-NAME owner record. It is likely that other WORK-FILE sets would be used for the conventional unstructured disk files maintained by a time-sharing system (e.g. source code files, object libraries, load modules, sequential data files and scratch files).

Specifically, within the context of the Michigan Terminal System, spooled files and <u>all</u> user disk files would be maintained as WORK-FILE sets. However, some file systems (e.g. under the UNIX operating system) support highly structured user directories and user files. Such facilities can be implemented by including record occurrences of type other than FILE-NAME as members of the USER-DIRECTORY set, and/or allowing FILE-NAME records to be owners of sets of type other than WORK-FILE. However, these extensions are not visible in the subschema of Figure A.11, where a user process views spooled files as simple, sequenced sets of

TEXT records.

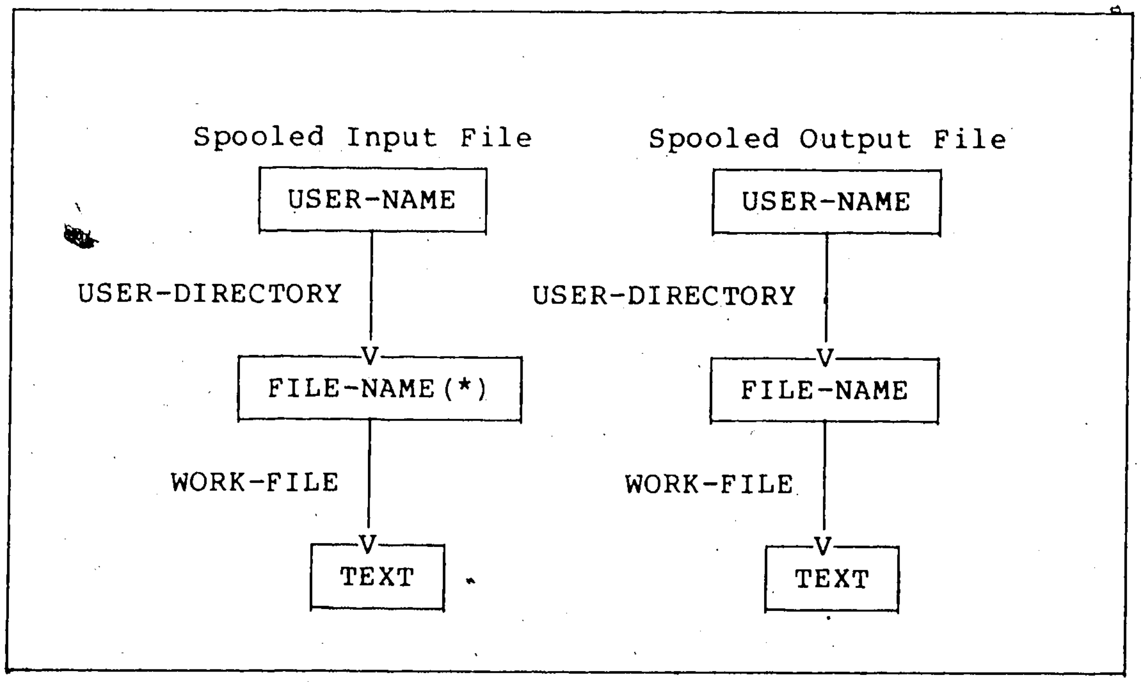

Figure A.11 Subschema for User Processes Using Spooled Files

Just as the Output Spooler is responsible for deleting the FILE-NAME record (and hence the WORK-FILE set) once the spooled file has been output to the physical device, someone must assume the responsiblity for deleting an input spooled file once the user process has 'finished' with it. To guarantee that this is done correctly, the task is not left to the user, rather the input-output module removes the set and ensures that the disk space can be re-used. This distinction between input and output spooled files is shown in Figure A.11, by appending the annototation '(*)' to the FILE-NAME record type for the input spooling subschema.

In their present form, the CODASYL DDL specifications do not support a 'delete after use' facility, however it would be relatively simple to include, given a clear definition of what constitutes having 'finished' with a set occurrence (e.g. the user process must close and release a set b[illegible] can be removed). Of course, the necessary checks [illegible] have to be included to ensure that the deletion did not [illegible]e place until an intervening 'checkpoint' had occurred -- otherwise the user process may not be restartable following a system failure or rollback.