NASA Contractor Report 178388

The Computational Structural Mechanics Testbed Architecture: Volume V - The Input-Output Manager DMGASP

Carlos A. Felippa

Lockheed Missiles and Space Company, Inc.
Palo Alto, California

Contract NAS1-18444

March 1989

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

Preface

The first three volumes of this five-volume set present a language called CLAMP, an acronym for **Command Language for Applied Mechanics Processors**. As the name suggests, CLAMP is designed to control the flow of execution of Processors written for NICE, the **Network of Interactive Computational Elements**, an integrated software system developed at Lockheed's Applied Mechanics Laboratory.

The syntax of CLAMP is largely based upon that of a 1969 command language called NIL (NOSTRA Input Language). The language is written in the form of free-field source command records. These records may reside on ordinary text files, be stored as global database text elements, or be directly typed at your terminal. These source commands are read and processed by an interpreter called CLIP, the **Command Language Interface Program**. The output of CLIP does not have meaning *per se*. The Processor that calls CLIP is responsible for translating the decoded commands into specific actions.

The ancestor of CLIP, LODREC, was patterned after the input languages of ATLAS and SAIL, two structural analysis codes that evolved at Boeing in the late 1960s. More modern language capabilities, notably command procedures and macrosymbols, have been strongly influenced by the Unix^{TM} operating system and the C programming language. The Unix "shell/kernel" concept, in fact, permeates the architecture of the NICE system, of which CLIP is a key component.

NIL and its original interpreter LODREC, which now constitutes the "kernel" of CLIP, has been put to extensive field testing for over a decade. In fact NIL has been the input language used by all application programs developed by the author since 1969 to 1979. (NIL also drives the relational data manager RIM developed by Boeing for NASA Langley Research Center.) During this period many features of varying degree of complexity were tried and about half of them discarded or replaced after extensive experimentation. CLAMP represents a significant enhancement of NIL, particularly as regards to directive processing, interface with database management facilities, and interprocessor control. The current version is therefore believed to be powerful, efficient, and easy to use, and well suited to interactive work.

Volume I (NASA CR-178384) presents the basic elements of the CLAMP language and is intended for all users. Volume II (NASA CR-178385), which covers CLIP directives, is intended for intermediate and advanced users. Volume III (NASA CR-178386) deals with the CLIP-Processor interface and related topics, and is meant only for Processor developers. Volume IV (NASA CR-178387) describes the Global Access Library (GAL) and is intended for all users. Volume V (NASA CR-178388) describes the low-level input/output

(I/O) routines.

This Manual describes a separable component of a multilevel data management system for the NICE network (NICE-DMS): the *input-output* (I/O) Manager. The I/O Manager functions as focal point for *direct-access I/O* and attendant auxiliary-storage administration activities initiated by its user program.

The *implementation* of the I/O Manager for NICE-DMS is called DMGASP. This is a package of FORTRAN 77 subroutines, which are supported by assembly language routines on some computers. Versions of DMGASP are available for operation on Univac 1100, CDC Cyber, VAX 11/780 and IBM 3000-series computers.

This document consists of several sections in which the technical scope and capabilities of DMGASP are discussed, and information required to make use of it in a *stand-alone configuration* is given. In addition, some material on auxiliary storage devices is collected to avoid references to scattered sources.

Warning: This Manual is *not* a tutorial document. It is intended only for advanced programmers, and as backup reference for maintenance of NICE-DMS. NICE processor developers not qualifying as I/O experts should limit themselves to the higher data management levels (*i.e.*, EZ-GAL), and avoid direct calls to the I/O Manager.

Contents

Contents

Contents

Contents

Contents

Tables

1
Introduction

§1.1 THE I/O MANAGER AND NICE

NICE (Network of Interactive Computational Elements) is a database-coupled, executive-less, integrated software system under development since 1980 at Lockheed's Applied Mechanics Laboratory. NICE consists of *architectural components* described in [1], and of computational elements called *processors*.

The NICE Data Management System (NICE-DMS) is one of three architectural components, the other two being processor-execution control and source-code maintenance. NICE-DMS implements advanced techniques for the administration of scientific databases. It is a *multilevel* system, the main components of which are shown in Figure 1.1. In the present document, we shall be concerned with the box labeled "I/O Manager". Instructions are given in Appendix A for compiling the NICE-DMS code on a VAX/VMS system. A glossary of commonly-used terms is given in Appendix B. Paged I/O performance comparisons are given in Appendix C. Several useful utilities are described in Appendix D. An index for this document is provided in Appendix E.

§1.1.1. Role of the I/O Manager

The most primitive level of NICE-DMS is the Input-Output Manager (IOM), which is implemented as a subroutine package called DMGASP. The IOM functions as a modular interface between operating-system software devoted to Input-Output (I/O) functions, and the higher levels of NICE-DMS.

These I/O functions pertain mainly to the use of *auxiliary storage* facilities such as direct-access mass storage (disks, drums), sequential-access storage (tapes, cassettes) and extended core storage. The present I/O Manager emphasizes direct-access devices, while sequential-access devices are being phased out.

REMARK 1.1

Readers familiar with database management literature will recognize DMGASP as the *access method* of NICE-DMS, *i.e.*, the software that maps a stored-record interface onto a physical-record interface.

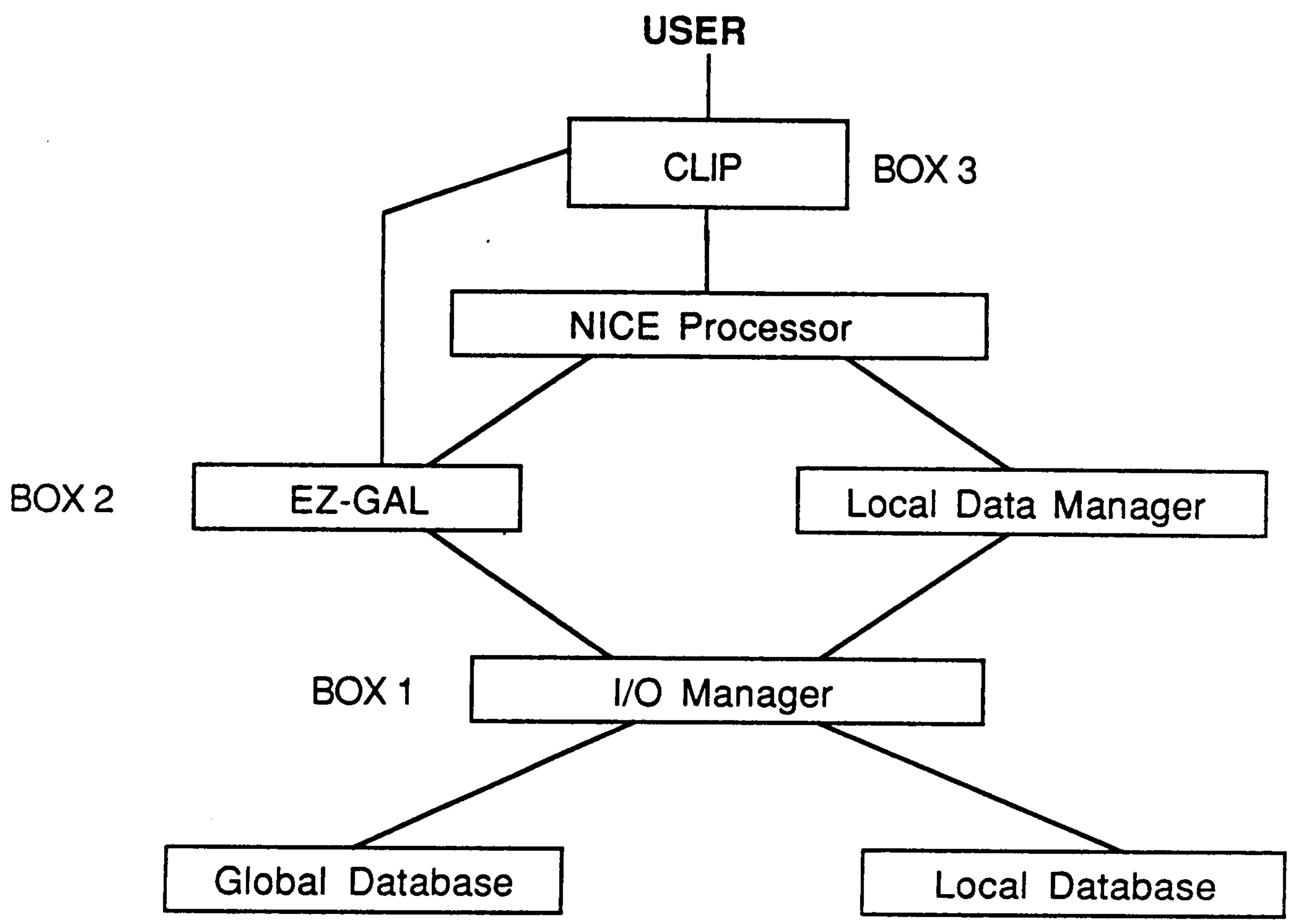


Figure 1.1. Configuration of NICE-DMS: Boxes 1, 2, and 3 are system-wide architectural components

§1.1.2. Stand-Alone Configuration

The higher levels of NICE-DMS, such as the global data manager [2] and the Command Language Interpreter Program CLIP [3,4,5], are not dealt with in this document. Instead, the configuration that readers should keep in mind is that of Figure 1.2. This displays only two software components: the I/O Manager (IOM) and the *user program.* The term "user program" means *all software external to the IOM and that calls it.* Or, to borrow a term from acoustics: the user program is the IOM's near-field.

For a NICE processor, the user program includes EZ-GAL and possibly the local data manager, as illustrated in Figure 1.1. The NICE processor "kernel" *never* calls the I/O Manager directly. Some computer programs may call all the I/O manager directly, in which case the application program *is* the user program.

The remaining subsections provide an overview of DMGASP and its interaction with typical storage facilities.

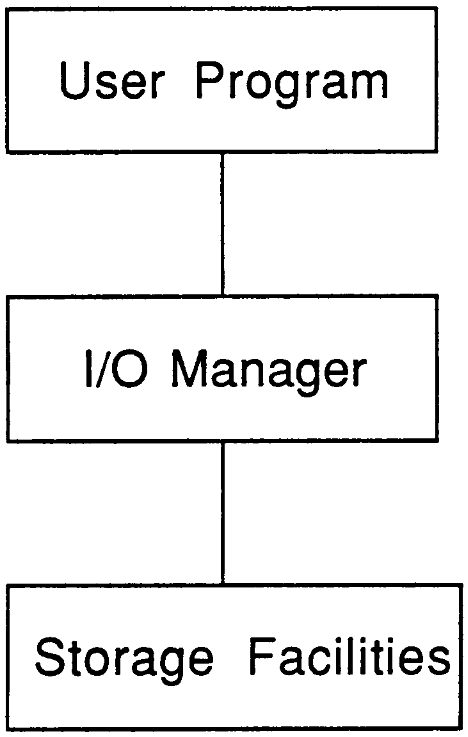


Figure 1.2. I/O Manager DMGASP Operating as a Stand-Alone Module

§1.2 STORAGE COMPONENTS

The present section deals with fairly standard material and serves primarily to introduce terminology. It should be ignored on first reading.

Storage is used to retain data and programs until they are needed during execution.

Storage facilities available at a computer installation consist of *physical storage devices* or briefly *devices*, such as main storage, disks and tapes. The I/O Manager further separates these facilities into *logical devices*, a concept that abstracts many hardware details and is taken up anew in §1.4. In the present section, *device* means physical storage device.

A storage device is divided into a finite set of components called *locations*. Each location can represent any one of a finite set of data *values*. These values are recorded and retrieved by *write* and *read* operations. A location is identified by its address.

One can characterize storage devices by the manner in which storage units can be accessed efficiently.

A *sequential-access device* consist of storage units which can only be accessed in sequential order. The device is positioned at its first location by a *rewind* operation. Read and write operations access the current location and position the device at its next location.

A storage device with *direct access* consists of storage units which can be accessed in arbitrary order by indexing. Most computers use two kinds of directly accessible stores:

1. *Main storage*, also called an *internal store*, gives very fast access to locations called *bytes* (or *words* in some computers). The storage medium is usually integrated circuits. The time required to access a location is independent of its physical position; this location process is called *random access*.
2. *Auxiliary storage*, also called a *backing store*, gives slower, direct access to storage units consisting of *blocks* of bytes. It is used to hold data or programs until computations need them in the internal store. The storage medium is often rotating magnetic surfaces (drums, disks). In a rotating storage device, a data block can only be accessed when the rotation of the medium brings it under an access head. This is called *cyclic access*.

Storage in present computers is usually provided by a *hierarchy* of devices of different types. Typically, a fast internal store of moderate capacity is backed by a slower, larger auxiliary store, which in turn is backed by a much larger but still slower file store.

The motivation for this variety of storage components is economic: the cost of storage is roughly proportional to the storage capacity and its access rate. It would be prohibitively expensive to maintain all user's programs and data permanently in main storage. Instead, users and operating systems try to distribute programs and data at various levels in the storage hierarchy according to their expected frequency of usage.

§1.3 SCOPE OF DMGASP

§1.3.1. Basic Operations

The I/O Manager DMGASP controls the use of storage facilities by its user program. Emphasis is placed on the management of *auxiliary* storage. Control is exercised through five basic operations:

1. *Open* a device (facility acquisition)

2. *Position* a device

3. *Read* a record from a device

4. *Write* a record on a device

5. *Close* a device (facility release)

These basic operations constituted the nucleus of DMGASP. Over the years they have been augmented with display, error-handling and supplemental operations.

REMARK 1.2
The open and close operations were called "declare" and "free" in earlier versions of this document. But open and close is now widely accepted terminology in I/O systems.

§1.3.2. FORTRAN and Block I/O Access Modes

Regarding read/write activities (items 3 and 4 of the previous list), two *file access methods* are possible on most computers:

FORTRAN I/O. Read and write operations are effected through FORTRAN READ and WRITE statements that operate on *direct-access* FORTRAN files. Some form of *buffering* by the FORTRAN I/O library is always involved, but this is beyond the control of DMGASP.

Block I/O. Data is moved directly in unbuffered "block" form from main to auxiliary storage and vice-versa, using calls to operating-system services.

Use of Block I/O normally results in considerable efficiency gains, especially for large records, but sacrifices portability.

REMARK 1.3
The original DMGASP provided only Block I/O. With the development of the FORTRAN 77 version, FORTRAN I/O appeared, and now constitutes the core of the "portable" version. This means that if the I/O Manager code is transported to a new machine, only FORTRAN I/O is made available. As time passes, a Block I/O capability may be eventually provided to make I/O more efficient.

REMARK 1.4

For versions that include both FORTRAN and Block I/O, the user has the choice of selecting one or the other for each particular device. The choice is done at the time the device is initiated. More details are provided in following sections.

REMARK 1.5

Access methods cannot be intermixed for a given device.

§1.3.3. Paged I/O

The present I/O Manager allows an alternative to the direct data transmission mode described in §1.3.2:

Paged I/O. Data transfer operations take into account the presence of a *page buffer pool* specified by the user program. This pool is shared by all devices open as paged. Pool resources are assigned on demand, and page exchange is based on a LRU (least recently used) scheme.

The presence of a page buffer may increase I/O efficiency when the transmission of many small but clustered records is involved. But the precise meaning of "small" and "clustered" is machine dependent. Quantitative data for a VAX system are provided in Appendix C.

Paged I/O may be performed on either Block I/O or FORTRAN I/O devices, although in practice it works best with the former (as Paged FORTRAN I/O in fact entails double buffering).

§1.4 DEVICE MANAGEMENT OVERVIEW

§1.4.1. Logical Devices

Storage facilities managed by DMGASP are separated into *logical devices* defined by the using program. Logical devices are referenced through *Logical Device Indices* (LDI). An LDI is an integer in the range 1 through MAXLDI, where MAXLDI is an adjustable internal parameter (presently 16). Throughout the remainder of this document, the term *device* is used in the sense of *logical device*, while *device number* means the corresponding LDI.

When the user program begins execution, all legal device indices are considered *undefined* or *inactive.*

Devices may be categorized according to their residence medium into *auxiliary storage* and *core* devices.

§1.4.2. Auxiliary Storage Devices

A *auxiliary storage* device is *opened* (declared, activated, assigned, attached) through the following procedure (see Figure 1.3).

1. The user program submits requests for storage facilities to the I/O Manager by supplying general characteristics of the desired equipment, a Logical Device Index, and (usually) a device name.
2. If the request is for a Block I/O device, the I/O Manager directly relays the request to the operating system. Assuming the request is granted, the assigned facilities are entered in the run's file directory with the device name (explicitly supplied or selected by default) used as identifier or *external file name* of the system file associated with the device. On some operating systems, the external file name is linked to an *internal file name*, which is subsequently used for requesting services such as record transfers.
3. For a FORTRAN I/O device, a FORTRAN *logical unit* serves as link between the LDI and the device name. The open request is submitted to the FORTRAN I/O library using an OPEN statement. If this is granted, the assigned facilities are entered in the run's file directory as in the previous case. All subsequent services are requested through the logical unit.

(a) Block I/O device with internal file name:

$$\begin{tikzcd}
\text{LDI} \arrow[r] & \text{Internal file name} \arrow[r] \arrow[d, leftrightarrow] & \text{Disk file} \\
& \text{External file name} &
\end{tikzcd}$$

(b) Block I/O device without internal file name:

$$\begin{tikzcd}
\text{LDI} \arrow[r] & \text{External file name} \arrow[r] & \text{Disk file}
\end{tikzcd}$$

(c) FORTRAN I/O device:

$$\begin{tikzcd}
\text{LDI} \arrow[r] & \text{Logical unit} \arrow[r] \arrow[d, leftrightarrow] & \text{Disk file} \\
& \text{External file name} &
\end{tikzcd}$$

Figure 1.3. Schematic representation of LDI-to-file concatenation for auxiliary storage device

A declared device is said to be *active*. Attributes of active devices are kept in a *logical device table* (LDT). Active devices can be positioned, written upon and read from by following the procedures outlined in §3.

An active device may be *closed* (deactivated, freed) at any time during the run. The released storage facilities are returned to the system. The device index is then considered inactive until a subsequent declaration, if any, attaches the LDI to other (or the same) facilities.

REMARK 1.6

The reader should not be unduly discouraged by the apparent complexity of the LDI-to-file concatenation process. Upon opening a device, all subsequent I/O Manager transactions are made through its LDI. The advantage of this is that the programmer is spared distracting hardware details, and may simply think of the LDI of an open device as a path to the physical storage.

REMARK 1.7

For everyday use of the I/O Manager, the programmer never "sees" internal file names or logical units, and indeed may forget that such things exist. These things become important when the IOM is moved to a different computer, or when the user-program developer worries about file name or logical unit clashing. To help on the latter subject, file-system characteristics of three computer systems on which the I/O Manager runs are reviewed in Sections 2.6-2.7.

§1.4.3. Core Devices

Devices may also reside on main storage, more specifically FORTRAN blank common. These are called *core devices*. The activation process does not involve file management, and thus it is considerably simpler.

§1.5 RESERVED SYMBOLS

Application programmers dealing with NICE-DMS in general and the I/O Manager in particular should take note of NICE-DMS global-symbol conventions in order to avoid name conflicts at link time.

Symbols that begin with the following letter combinations are used for naming entry points:

DM, GM, IO, LM

More specifically: externally callable subroutines at the IOM and EZ-GAL levels are named **DM***xxxx* and **GM***xxxx*, respectively. Externally callable integer functions at both levels are named **LM***xxxx*. And entry points "hidden" within the I/O Manager are named **IO***xxxx*.

Symbols that begin with the letter combinations

CDM, CIO, CGM

are used for naming internal common blocks.

2
Device Management

§2.1 THE PHYSICAL ORGANIZATION OF DATA

Effective utilization of the I/O Manager demands some understanding of the physical organization of data in computer storage. The required knowledge basically reduces to two subjects: (a) the manner in which the data are stored, and (b) the manner in which the data are accessed.

This knowledge is necessary because the I/O manager operates very close to the actual representation of data on the hardware. A less detailed knowledge is required for programmers that deal only with the Global Database Manager EZ-GAL, since the GDM conceals many hardware details through a logical-to-physical mapping process.

This Section covers aspects of device management that are necessary for effective understanding of the IOM operations described in Sections 3 to 6. Inasmuch as the present implementation emphasizes direct-access devices, these are the only ones covered.

§2.2 STRUCTURE OF DIRECT-ACCESS DEVICES

§2.2.1. Physical Record Units

A direct-access device may be viewed as a linear array of *physical record units* (PRU). A PRU is defined as *a storage unit that may be read or written without need of accessing or modifying adjacent PRUs.* Or, to put in another way: read and write operations can start only at a PRU boundary.

Each PRUs is identified by its associated sequence number, *counting from zero*, as illustrated in Figure 2.1.

```
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
```

Figure 2.1. Direct-Access Device as PRU Stream

The sequence number is the PRU *location.* To *address* a direct-access device, one specifies the PRU location at which data are to be stored or retrieved in the next write-record or read-record operation. This procedure is called *device positioning*; more about it in §2.3.

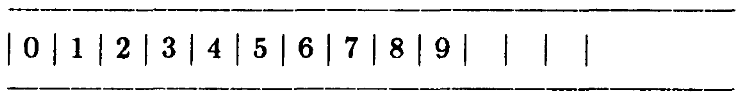
The *length* or *extent* of a direct access device is the index of the highest PRU written. This PRU is called the *end of information* (EOI). The PRU that follows the EOI is called the *next free location.*

It is seen that the concept of a PRU naturally introduces read and write constraints. These are further elaborated upon in §2.3.2.

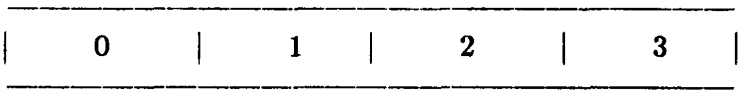
So far, all of this seems straightforward. But complications arise from the fact that the PRU *size* varies according to the level from which one looks at the direct access device. More specifically, the I/O manager has generally to know about *three* PRU sizes: external, internal, and hardware, as depicted in Figure 2.2.

This PRU hierarchy exists because of conflicting requirements. A small PRU size optimizes storage utilization and simplifies addressing, but is detrimental to I/O performance. These requirements can be balanced by presenting a small PRU size to the user, while internally mapping transfers of records expressed in the smaller units into coarser block transfers.

(a) **External PRU** at the user/IOM interface:



(b) **Internal PRU** at the IOM/OS interface:



(c) **Hardware PRU** at the OS/device-driver interface:

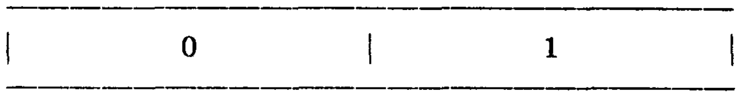


Figure 2.2. PRU "Granularity" of Direct-Access Devices according to Software Level

§2.2.2. The View from Outside

The IOM user sees a direct-access device as an array of *external PRUs.* An external PRU is one or more words, and is specified — explicitly or implicitly — when the device is opened.

If the external PRU size is exactly one word, the device is said to be *word addressable.* This is the default setting; *i.e.*, if nothing specific is said about external PRU size at device open time.

If the external PRU is one byte, the device is said to be *byte addressable.* The present I/O Manager does not support byte-addressable devices; however, such addressability is simulated by the Global Data Manager EZ-GAL for certain character-record operations.

In some old-fashioned file structures, notably those used by the DAL data management system, the external PRU size is equal to a disk sector (defined in §2.2.4). These are called *sector addressable* files.

The external PRU size is used as the addressing unit for all device positioning operations. Thus, this level is the only one that most IOM users need to know or worry about.

§2.2.3. The View from Inside

Deep within IOM, a different storage unit is used: the *internal PRU.* This is the block size used for physical I/O requests to the operating system. It is also the size of internal IOM buffers that take care of the internal/external PRU alignments.

The selection of an internal PRU varies according to the storage medium and device access method.

Disk-Resident Block-I/O Devices. The internal PRU size coincides with the hardware PRU size; so in this case there are effectively only two PRU sizes.

FORTRAN I/O Devices. The internal PRU size is the "record length size" parameter of FORTRAN Direct-Access Files. This size is set according to a device-type descriptor when the device is opened (see §2.5), and either divides exactly the hardware PRU size, or is an exact multiple of it.

Core-Resident Device. The internal PRU size is the same as the external PRU, *i.e.*, one word.

REMARK 2.1

Knowledge about the internal PRU size is useful in special situations. One of these special situations would involve a FORTRAN I/O file created by the I/O manager and which is to be read directly by a DMGASP-less program.

§2.2.4. Hardware PRU Size

The hardware PRU size is directly related to equipment characteristics. Hardware PRU sizes for *disk-resident* devices vary from 4000 to 8000 bits. Some examples:

Univac 1100: 112 four-byte words (1 byte = 9 bits)
CDC Cyber: 64 ten-byte words (1 byte = 6 bits)
VAX 11/780: 512 bytes (1 byte = 8 bits)
IBM 370: variable; commonly 800 bytes (1 byte = 8 bits)

For disk and drum devices, a hardware PRU is often identified with a *sector*, which is the smallest addressable segmentation of a disk track. But I/O system simulation may make a sector appear smaller. For example, Univac 1100 disk peripherals have a sector size of only 28 words.

The hardware PRU of core-resident devices is either one word or one byte, depending on the addressibility characteristics of the computer.

§2.3 RECORDS

The I/O Manager recognizes only one data object: the *IOM record*, or record for short.

§2.3.1. Definition

An IOM record is an *array of words* characterized by physical adjacency, and which can be read or written with a *single* call to the I/O Manager.

An IOM record can reside at any word-aligned location of main storage; this includes of course core devices. Auxiliary storage records must reside in logical devices accessible to the I/O Manager, and respect external-PRU alignments as explained in §2.3.2.

There are *no* end-of-records marks. Application programmers that work at the IOM level are free to do imaginative (and dangerous) things like leaving "holes" between records, rewriting records with smaller or larger ones, or reading many adjacent records as one.

REMARK 2.2

The key terms in the definition are *array* and *word*. Note that the external PRU size is not mentioned in the definition, since it only affects alignment constraints (cf. §2.3.2).

REMARK 2.3

As far as record transmission is concerned, the IOM does very little preliminary checking before passing the request to the operating system or the FORTRAN I/O library. On read operations, it tests whether the record falls within the current device extent (PRUs 0 through EOI). It is even more permissive on write operations: it simply checks whether the record would not go beyond the device capacity limit. No diagnostic is given for trying to read from undefined areas in the middle of a device; diagnostics will be given by the operating system.

REMARK 2.4

Previous IOM versions also kept track of *physical files* on magnetic tape devices. With the disappearance of sequential-access devices, the concept of physical file becomes unnecessary.

REMARK 2.5

An IOM record occupies an intermediate position between a *logical record* as seen by the application program ("get me a stiffness record") and a *physical record* as seen by the I/O system ("ship these data blocks to so-and-so disk track").

§2.3.2. PRU Alignment

Records on auxiliary storage devices are *left-aligned* with external PRU boundaries, as illustrated in Figure 2.3.

```
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |    |    |    |    |    |
          =Record A=          ==Record B===   =Record C==
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |    |    |    |    |    |
```

Figure 2.3. Record Alignment Constraints
(space between vertical bars = external PRU)

If the device is not word-addressable, records generally do not fill the last PRU, as in the case of Record A in Figure 2.3. Users of such devices should be then aware of the *interrecord gap* problem, which is discussed in §2.3.4.

§2.3.3. Device Positioning

Positioning a device means specifying the PRU index at which the next record read or write is to take place. Positioning may be *explicit* or *implicit.*

To read Record A shown in Figure 2.3, the device is *explicitly* positioned at PRU number 2 (the *third* PRU) and a read request issued. On successful completion, the device is *implicitly* positioned at PRU number 6, which is the PRU that immediately follows record A. To read Record B, the device is positioned at PRU number 7 — another explicit positioning — and a read issued. On return, the device is implicitly positioned at PRU number 10, which is where Record C begins, so this record be may read without pre-positioning.

Explicit positioning is performed by calling the I/O Manager and telling it the PRU index. This operation is different from a read or write operation; positioning-and-read or positioning-and-write are *not* a single operation at the IOM level. Explicit positioning does not result in any mechanical operation (as occurred when the old I/O Manager positioned a magnetic tape): the PRU index is simply stored in the Logical Device Table.

Implicit positioning occurs as a side result of read and write operations, and may be exploited to read or write records *sequentially.*

The PRU "positioning" index maintained by the IOM for each device is called the *Current Device Location*, a state parameter often abbreviated to CDLOC. A device for which CDLOC = 0 is said to be *rewound* (= positioned at its beginning). A just-open device is always placed in the rewound state.

§2.3.4. Interrecord Gaps

Advanced users ought to be aware of the "gap" problem that plagues devices that are not word-addressable.

Suppose that the external PRU size of a device is 16 words. The device is positioned at its beginning (PRU number 0) and a 32-word Record A, which covers exactly two PRUs, written. Then the device is repositioned at the beginning, and a 24-word Record B (one and a half PRUs) written. Finally, reposition again to PRU number 0, and read 32 words. What do you expect to find in the last 8 words?

The answer is that *nothing* should be expected. On some computers, you will find the last 8 words of Record A intact. But on others you will find meaningless data — for example system-buffer "leftovers" from an unrelated process. Much depends on what happens between the two record writes. This is the infamous *interrecord gap* phenomenon: trailing locations on the last PRU written should be viewed as *undefined.*

If records were read exactly as they are written, the presence of interrecord gaps would be harmless. But IOM programmers typically will tend to take risks. Presetting the entire device extent to zero, which some inexperienced programmers have suggested, is not necessarily a solution on *every* computer.

If word-addressable devices are used, no interrecord gaps can occur, and the problem goes away. This is one strong reason for using word addressing as standard, despite a slight loss in I/O performance.

§2.4 RECORD DYNAMICS AND FILE INTEGRITY

An understanding of what actually happens to records on their way from source to destination is important for issues such as insuring data integrity under abnormal run terminations. The discussion below is limited to *auxiliary storage* devices resident on disk files.

§2.4.1. Unpaged Block I/O

Unpaged Block I/O is the easiest to understand. Records are moved directly between the user program work area and the disk file in which the logical device resides, as sketched in Figure 2.4.

Work area ⟨==============⟩ Disk file

Figure 2.4. Record Transmission in Unpaged Block-I/O Device

Notice the absence of *visible* stops or detours. (Records may be temporarily broken up within IOM "alignment buffers" that take care of external/internal PRU size mismatches, but *are not held* there.)

If the user program terminates abnormally, records previously written (without error) with Unpaged Block I/O are guaranteed to be in the disk file. They might be inaccessible, but they are there.

§2.4.2. Unpaged FORTRAN I/O

In FORTRAN I/O devices, records go through the buffering system of the FORTRAN Run-Time Library (FRTL), as shown in Figure 2.5. The buffer structure is machine-dependent. Some FRTL implementations assign distinct buffers to each logical unit, while others maintain a buffer pool for all units.

Work area ⟨=====FRTL Buffers =====⟩ Disk File

Figure 2.5. Record Transmission in Unpaged FORTRAN I/O Device

These buffers are beyond the reach of both I/O Manager and user program. A close-device operation writes the contents of the buffers to the external device (this process is referred to as flushing the buffers) and guarantees the integrity of a file that is to survive execution. But what if the run suddenly aborts?

What happens depends on how good the FORTRAN I/O system is. On a decent system, write-buffers are automatically flushed and open files closed. But under some operating systems, consider yourself lucky if you can salvage anything from the program abort.

§2.4.3. Paged Block I/O

For Paged I/O devices, there are more complications. Consider first the (most common) case of Paged Block I/O. Interposed between the user's work area and the disk file is a Page Buffer Pool (PBP), as shown in Figure 2.6.

Work area ⟨===== Page Buffer Pool =====⟩ Disk File

Figure 2.6. Record Transmission in Paged Block-I/O Device

The PBP is *shared* by all Paged devices, regardless of whether their file access method is Block or FORTRAN I/O. The PBP is subdivided into *pages* of equal size. The motivating idea is to try to reduce the number of physical data transfers when reading or writing clusters of small records, since the I/O Manager recognizes that data wanted by the user program are in the PBP. (This is not possible in the configuration of Figure 2.5, since FORTRAN buffers are beyond IOM's control.)

Explicit closing of a Paged device that has been written during the run insures file integrity. The main concern with such devices is that, should the run suddenly abort, records or record segments are left in the PBP. To alleviate this problem, the I/O Manager provides a "flush-PBP" service (§4.3), which writes all *modified* pages to the appropriate device(s), without closing the device(s). Judicious use of this service for critical files that are to survive the run is highly recommended.

§2.4.4. Paged FORTRAN I/O

If a Paged device uses FORTRAN I/O, there are now *two* buffers being manipulated, as shown in Figure 2.7.

Work area ⟨=== PBP === FRTL Buffers ===⟩ Disk File

Figur9e 2.7. Record Transmission in Paged FORTRAN-I/O Device

For these devices, one has to be doubly cautious to insure file integrity on abnormal termination. Periodic PBP flushing should be complemented by robust error-termination procedures.

§2.5 DEVICE DESCRIPTORS

Attributes of logical devices are identified at the time the device is opened ("open time") by four integer parameters known as *device descriptors.* They are:

Device Type Index (TYPEX). Characterizes device residence medium and file access method; for example: disk-file or core residence, block I/O or FORTRAN I/O.

Device Options Index (OPTX). Characterizes device permanency and use attributes; for example: existing or new, write-allowed or read-only.

Device Capacity (LIMIT). Characterizes the maximum extent to which a newly created device may expand.

External PRU size (XPRU). Characterizes the external PRU size of auxiliary storage devices; also used to flag Paged I/O.

These four parameters are described in the following subsections.

§2.5.1. Device Type Index

The device type index (TYPEX) characterizes *intrinsic* attributes, *i.e.,* those retained throughout the lifetime of the device. The index may vary from -4 to +5, and the associated attributes are as specified in Table 2.1.

Block I/O Devices. Device types 0 and 1 are associated with disk-resident Block I/O devices. In the present implementation, 0 and 1 are in fact equivalent. A distinction was made, however, in the old Univac version and may reappear in future versions, so use of type 1 should be avoided.

> **Warning:** Types 0 and 1 are available only if Block I/O has been implemented for a specific machine *and* operating system.

FORTRAN I/O Devices. TYPEX values of 3 and 4 correspond to direct-access devices implemented through FORTRAN 77 direct-access files. As this type of I/O is machine-independent, it is available on any computer equipped with a FORTRAN 77 compiler. Block I/O should be preferred, however, if available because of its superior performance.

The only difference between types 3 and 4 is the choice of internal PRU, as specified by Table 2.2. If nothing is known beforehand about the record size mix that the device will handle, chose TYPEX = 3. If it is known that the device is to handle only fairly large records (of the order of 500 words or more), or if the device will be Paged, select TYPEX = 4. Be sure also to read Remark 2.7.

Table 2.1
The Device Type Index (TYPEX)

TYPEX	*Device type*
0	Block I/O on disk
1	Block I/O on disk (reserved for future use; see Remark 2.8)
2	Block I/O on drum (obsolete)
3,4	FORTRAN direct access
5	"Core device" in blank common
-4 to -1	FORTRAN sequential access (obsolete, see Remark 2.9)

Table 2.2
Internal PRU for FORTRAN I/O Devices

Machine	*TYPEX*	*Internal PRU size in words*
Univac	3	28
	4	112
CDC	3	64
	4	256
VAX and IBM	3	32
	4	128

Core Devices. TYPEX = 5 specifies a "core device": a word-addressable device resident in main storage, more specifically blank common. This is of course a *scratch* storage facility, which disappears when the user program stops executing. Core devices find application in fancy local data management, but they are not selected by everyone.

REMARK 2.6

Type indices should not be intermixed during the lifetime of a file-resident device. For example, suppose a permanent-file device is created with TYPEX = 3. When the device is opened again as an "old" file, index TYPEX = 3 should be specified, and no other. The only exception is that TYPEX = 0 might be used to open an existing FORTRAN I/O device and *read* (never write) from it, but this subterfuge does not work on all computers. The I/O Manager *does not* check for TYPEX-mixups when you open a device (it has no easy way of knowing); but EZ-GAL does check for it, as the value of TYPEX at the time of device creation is saved in the library file header.

REMARK 2.7

In previous IOM versions, TYPEX was called the *equipment code index*, as it truly characterized the physical residence medium: drum, disk, tape, extended core. With the gradual disappearance of drums and tapes, and a trend to get away from hardware details, the index now characterizes primarily the device access method.

REMARK 2.8

Device type 2 identified Block I/O on drum devices connected to old Univac 1107 and 1108 computers. As drums have disappeared from the computer scene, TYPEX = 2 has no present use, but is kept in reserve for possible applications in the future.

REMARK 2.9

Negative TYPEX values were used in previous versions for handling tape-resident sequential-access devices. In the present version, they can be used for opening and closing sequential-access FORTRAN files, but such files cannot be positioned, read, or written through the IOM. These device types exist only to support old FORTRAN 66 programs. (FORTRAN 77 programs can dynamically open and close such files through the OPEN and CLOSE statements, which are now standard part of the language.)

§2.5.2 Device Options Index

The options index OPTX is an integer in the range -6 to +12 that characterizes device permanency and accessibility attributes. For example: is this a new, old or scratch device? are writes allowed? etc.

The most commonly used values of OPTX are the machine-independent values listed in Table 2.3. These are applicable to devices resident in auxiliary storage. For core-resident devices (TYPEX = 5), OPTX = 0 is automatically assumed.

Less common values of OPTX, which have meaning only on specific machines, are listed in Table 2.4.

§2.5.3 Device Capacity Limit (LIMIT)

The device-capacity limit parameter (LIMIT) specifies the maximum number of words a device may hold. This specification has effect only on newly created devices. Default values are given in Table 2.5.

On Univac, the LIMIT specification is used in the construction of the @ASG-file specification for Block I/O devices, which must specify a maximum number of disk tracks. On other computers, LIMIT is simply used as a safeguard against rogue expansion of disk files attributable to user-program errors.

§2.5.4. External PRU Size Parameter

A specific external PRU size (XPRU) in words may be specified at device open time. A zero value requests the IOM default, which is *word-addressing* (XPRU=1).

The default setting is highly recommended (read §2.3.4). PRU sizes other than the default should be specified only if there is a very good reason for it; for example, to read Block I/O files created by another program that uses XPRU > 1 (*e.g.,* DAL files).

There are certain constraints on the selection of multiword XPRU. The external PRU size *must divide exactly* the internal PRU size (to find out about the latter, consult Sections 2.1.3-2.1.4). This rule includes the limit case in which the two sizes coincide, but the external PRU size must never exceed the internal PRU size. For example, suppose that the internal PRU size is 128 words. Then the only legal external PRU are the powers 2^n, where n is 0 to 7.

§2.5.5. Logical Units

For FORTRAN I/O devices (TYPEX $=$ 3 and 4), the I/O Manager associates the LDI with a FORTRAN logical unit at the time the device is opened. From then on, the logical unit is used to request read, write and close services to the FORTRAN Run-Time Library. The IOM programmer never deals with the logical unit, however, only with the LDI.

The correspondence between LDI and logical units is maintained in an internal table, which is initialized at compile time using a DATA statement. Table 2.6 shows a typical correspondence table used on the VAX version. Note that units 5 and 6 are skipped as these are commonly connected to the line-reader and system-print files, respectively.

The default correspondence is not only machine-dependent but site-dependent. The latter becomes necessary on operating systems such as Univac's Exec-1100, under which certain unit numbers are reserved for specific function at the installation's discretion. Changing the default correspondence table involves a recompilation of DMGASP. But the table can also be changed at run time (usually once and for all at user-program start) through entry point DMUNIT (§4.9).

The NICE programmer that plans to use FORTRAN I/O devices should be generally aware of the following:

1. Try to avoid non-IOM use of units 1-4, 7-20. Use of units 5-6 for read-print is safe.

Table 2.3
Machine-Independent Values of OPTX

OPTX	*Options*
0	Open scratch device
3	Open existing device as read-only
4	Open existing device allowing writes
6	Open new device and catalog as permanent file (public on Univac)
-5	As OPTX = 3 if file exists, otherwise 6
-6	As OPTX = 4 if file exists, otherwise 6

Table 2.4
Machine-Dependent Values of OPTX

OPTX	*Options*	*Machine*
2	Link LDI to the file already assigned to run	Univac
5	As OPTX = 6, but catalog the file as private	Univac, CDC
7	As OPTX = 6, but catalog the file as private read-only	Univac
8	As OPTX = 6, but catalog the file as public read-only	Univac
9-12	As OPTX = 5-8, respectively, but do not catalog the file if the run ends abnormally	Univac

Table 2.5
Default Device Capacity

Device Type	*Computer*	*Default LIMIT*
Disk-resident	CDC	2^{24} words
	IBM	6,400,000 words
	Univac	1024 tracks (1 track = 1728 words)
	VAX	6,400,000 words
Core-resident	All	20,000 words

Table 2.6
Default LDI-Logical Unit Correspondence (VAX Version)

LDI	*Logical Unit*
1-4	1-4
5-16	7-18

2. If problems with units 1-20 develop, you may have to use DMUNIT. The default LDI-unit table may be seen by calling DMSTAT (§3.6) with an 'LDTF' argument, or using CLIP directive *LDT/F.

3. If you also use CLIP, avoid units 30-40.

§2.5.6. The File-Length Recovery Problem

The *length* or *extent* of a direct-access device was defined in §2.2.1 as the index of the highest PRU written (the end-of-information, or EOI). The PRU that follows the EOI is called the *next free location*, or NEXT.

The length attribute is important for devices resident on permanent disk files that are to be reopened and *extended* with more records. Imagine, for example, that you want to append records. If the EOI is not accurately known, a big gap may appear in the middle of the file, or, even worse, existing data may be inadvertently written over.

But the seemingly trivial matter of retrieving the exact length from the operating system when an existing file is reopened turns out to be surprisingly difficult. This is due to a combination of minor problems, which together produces a big problem.

Lack of Resolution. With the possible exception of Unix, no operating system maintains file-length information down to external-PRU "granularity". For example, Univac's Exec-1100 will tell you only "the highest track written". Since one track is 1782 words, this is three orders of magnitude too coarse for a word-addressable device.

Machine Dependency. Retrieval of even the coarse size information is by no means easy. Procedures vary from system to system, and use of convoluted assembly-language code is often required. There are some systems *(e.g.,* CDC's NOS) where access to this information is virtually impossible: their manuals won't tell you where it is.

A Gaping Hole in FORTRAN 77. Although FORTRAN-77 I/O is, on the whole, a considerable improvement over FORTRAN-66 I/O, the ANSI standard inexplicably passes over an important point: there is *no* way to inquire how big a direct-access file is. (For a sequential-access file, at least one can read until encountering the EOF, but direct-access files do not have EOF marks.)

The only clean solution to this vexing problem is to keep file extent information (EOI or NEXT) *in the file itself.* The file extent information must be stored near the file start so that it may be read at open time. The first word of the file (word 0) emerges as a sensible choice. This is in fact the scheme used by the global data manager EZ-GAL for word-addressable GAL files and sector-addressable DAL files.

By design, the I/O Manager *assumes nothing* about device contents. It follows that the process of retrieving the extent data and transferring it to the I/O manager must be done by the user program. Entry point DMNEXT (§4.5) is provided for this purpose.

§2.6 FILE SYSTEMS

Auxiliary storage devices reside on *system files.* Thus, the IOM user is expected to be aware of the peculiarities of the file system used by the host operating system.

In particular, the programmer should be aware of the conceptual distinction between *internal file names, external file names, and logical units.* This distinction exists, in one form or another, in all operating systems.

1. Each operating system has its own jargon for describing system actions: there is no standard terminology.

2. Each operating system has its own capabilities and limitations: there are no standard functions.

3. Some of the things we would like to do can't be done easily on some operating systems: there are no perfect systems.

When talking about file conventions, one must therefore carefully specify the host operating system.

REMARK 2.10

The material that follows is "refresher" material that need not be covered on first reading.

§2.6.1. File Conventions on Univac

Under Univac's Exec-1100 system, a file can be referenced by two names: external and internal.

The *external file name* is the one under which facilities for file residence are requested from the operating system, and also the identifier kept in the master file directory in the case of a permanent file (*catalogued file* in Univac terminology). External file names of permanent files must be unique across an installation.

The *internal file name* is the (usually short) identifier through which the file is referenced by a running program; these identifiers need be unique only for a specific run.

The two names: external and internal, may be connected through a @USE control statement [6] issued before or during program execution. If no @USE statement is issued, the names coalesce.

For a FORTRAN I/O statement such as

```
WRITE (14) (A(J),J=1,N)
```

14 is known as the *logical unit number*, which is simply a pointer to a table of internal file names maintained by the run-time FORTRAN I/O library. On Univac, the internal file name associated with unit 14 is

'14'

that is, the character-encoded representation of the unit number (left-justified with blank-fill).

§2.6.2. File Conventions on CDC

On CDC operating systems such as SCOPE [7] and NOS [8] the two terms *logical file name* (LFN) and *permanent file name* (PFN) are used in the sense of internal and external file name, respectively.

All files attached to a run, whether temporary (*local* files in CDC terminology) or permanent, are referenced by a logical file name, but only permanent files have a permanent file name. (Thus, for local files one may think of the LFN as functioning as both internal and external file names.)

Linking of local and permanent file names is accomplished by a potpourri of operating system commands, such as CATALOG, ATTACH, GET and SAVE. These operations are not only file-status dependent (*i.e.,* vary according to whether the file is new or already exists), but often change names from SCOPE to NOS. There is no clean USE statement as in Univac. Also, many CDC installations have unique local conventions designed to enhance job security.

A FORTRAN logical unit number such as 14 is conventionally associated with the *logical* file name

'TAPE14'

§2.6.3. File Conventions on VAX

The VAX/VMS operating system [9] offers a clean set of file conventions, much better than either Univac or CDC.

External file names identify *physical devices* belonging to a "file owner", while internal file names are known as *logical device names.* (This bears no relation to CDC terminology.) The two names may be linked through an ASSIGN statement [9].

A FORTRAN logical unit such as 14 is identified with the logical name

'FOR014.DAT'

Significant differences with Univac and CDC are:

1. The ASSIGN may refer to only components of the file physical device identifier; for example, the directory name.

2. All files are permanent, *i.e., retained in the user's directories*, unless *explicitly* declared as "scratch". This feature is characteristic of interactive operating systems. By way of contrast, Univac and CDC permanent files must be explicitly declared as such.

§2.6.4. File Conventions Summary

File conventions are summarized in Table 2.7.

Table 2.7
File-Systems Terminology Summary

Univac	CDC	VAX
External filename	Permanent filename	Physical device name
Internal filename	Logical filename	Logical device name
Temporary file	Local file	Scratch file
Catalogued file	Permanent file	Directory file
USE	Numerous ways	ASSIGN
ASG	ATTACH	CREATE, OPEN

§2.7 EXTERNAL DEVICE NAMES

Logical devices are identified at open time by *external device names*. The name may be *explicitly* supplied by the user, or be *implicitly* selected by the IOM if the user supplies a blank name. Implicit naming rules are shown in Table 2.8.

General recommendations regarding explicit versus implicit device naming are:

1. The user should specify the name for an auxiliary-storage device that resides, or will reside, on a *permanent file*.

2. The user should let the IOM pick up the name if the device is to reside on a *scratch file* or core storage.

For devices resident on auxiliary storage (usually disk), the external device name is either the *external file name*, or contains that name in some fashion. Rules to this effect are elaborated upon in subsequent subsections.

§2.7.1. External Device Names on Univac

The most general form of the external device name on Univac is

Qualifier*Filename(cycle)

where the Qualifier and cycle parts are optional.

The total length of the device name string is restricted to 24 characters. Qualifier and Filename may be of 1 to 12 characters in length, and the characters may include any combination of letters A-Z, digits 0-9, dollar sign, and dash. Default qualifiers are installation dependent; consult local manuals if in doubt. The file cycle is an integer in the range 1 through 63 and is rarely used.

Examples of legal Univac external device names:

QP35*RESPON   DT$05*FORM-2-12-82

REMARK 2.11

There is no need to include an ending period; but if given, it is treated as a file name terminator.

REMARK 2.12

Read and/or write key specifications are not permitted (they make little sense for data files, anyway).

§2.7.2. External Device Names on CDC

External device names of Univac-like form:

Qualifier*Filename(Cycle)

Table 2.8
Default External Device Names

Case	*Computer*	*Default Name*
LDI never activated	CDC	'TAPE*xx*', where *xx* is a logical unit number
	IBM	Presently same as VAX
	Univac	'UNIT*xx*', where *xx* is a logical unit number
	VAX	'FOR0*xx*', where *xx* is a logical unit number
LDI previously activated	all	Previous name

are accepted on the CDC version for *Block I/O* auxiliary storage devices resident on *permanent files.*

On the SCOPE operating system (also in NOS/BE), the total length of the external device name is restricted to 20 characters. The qualifier is interpreted as the *user's ID*, and the filename as the *permanent file name* (PFN). Qualifier and filename are restricted to a maximum of 8 and 7 characters, respectively. Only alphanumerics may be used, *i.e.*, letters A-Z and digits 0-9. Restrictions on user's ID are installation-dependent; consult local system documentation as appropriate. File cycles are frequently used on SCOPE. If the cycle specification is omitted, the highest catalogued cycle is assumed if attaching an existing permanent file, or the highest cycle plus one if cataloguing a new permanent file. (Many CDC installations restrict the number of simultaneously catalogued cycles to five.)

On NOS systems, previous constraints on length and legal characters apply, with the following additional restrictions. The qualifier is interpreted as the *user's catalog number*, which is fixed for each user and assigned by the installation. The file cycle specification is ignored, as NOS files have no cycle numbers.

For *local files* of any type, or for FORTRAN I/O devices, only the filename part is allowed. This stems from restrictions in the OPEN statement of CDC's FORTRAN 77 (FTN5) compiler. See Remark 2.14 for ways of circumventing problems caused by this restriction.

Of the two example Univac device names (§2.7.1), the first one:

QP35*RESPON

is also acceptable as a Block I/O device on SCOPE if QP35 is a legal user's ID (on most installations it will be). On NOS, QP35 must be the user's catalog number (or the catalog number of another user to whom the file belongs). The second example name is illegal because of the presence of dollar sign and dash characters; moreover, it exceeds the seven-character limit for the file name.

REMARK 2.13

To manipulate permanent files created with *FORTRAN I/O*, a two-stage process involving control cards is inevitable. Existing permanent files that are to be opened by the IOM should be attached before the run, then the local file name used as external device name. Created files that are to be cataloged as permanent should be cataloged on SCOPE after the run, or predefined on NOS before the run.

§2.7.3. External Device Names on VAX

On VAX computers, the IOM user has the option of using either Univac-like device names, or VAX/VMS names. The former are internally converted to the latter.

The name conversion process is best illustrated through examples. Consider the Univac-like external device name:

PR*RESPONSESDAT(8)

The I/O Manager converts this to

```
PR:RESPONSES.DAT;8
```

which is a legal VAX/VMS file identifier. The 12-character Univac-like file name, RESPONSESDAT, is split into a VAX file name, RESPONSES, and a file extension, DAT, because VAX file names are restricted to 9 characters. The Univac cycle specification becomes a VAX/VMS file version number. The most difficult to grasp is the qualifier transformation. On the VAX, PR is assumed to be the *logical name* of the directory to which the file belongs; this logical name must be declared (explicitly or through the LOGIN.COM file) by an ASSIGN command. For example, assume that the directory in question is [FELIPPA.NICE.SHOCK]; then

```
$ ASSIGN [FELIPPA.NICE.SHOCK] PR:
```

effectively links PR to that directory. If file RESPONSES.DAT;8 is created by the IOM, it will then appear in [FELIPPA.NICE.SHOCK].

In the frequent case where the file belongs to the default directory the qualifier may be omitted.

Of course the IOM also accepts standard VAX/VMS file names. For example,

```
PR:RESPONSES.DAT;8
[FELIPPA.SKY]SOLVER.DAT
```

The total length of the external device name string is restricted to 48 characters.

Explicit version numbers are rarely needed, because VAX/VMS works very much like CDC's SCOPE. Defaults are the highest version number for an existing file, or the highest version number plus one for a new file.

VAX/VMS directory names and file names can only have alphanumeric characters. Thus Univac-minded users are advised to leave out dollar signs and dashes.

THIS PAGE LEFT BLANK INTENTIONALLY.

3
Basic Operations

The basic operations provided by the I/O Manager DMGASP are: open device, close device, position device, write record to device, read record from device, and list state information. Entry points to perform these operations are listed in Table 3.1, and described in Sections 3.1-3.6.

Alternate entry points are offered for the open, close, position, write, and read operations. Three-argument entry points of the form DMxAST (where x = D,F,P,W,R) are compatible with previous IOM versions. The newer entry points DMOPEN, DMCLOS, ... etc., have the same first three arguments but include a trailing TRACE argument.

REMARK 3.1

The global manager EZ-GAL calls the four-argument entry points if its MSC (Master Source Code) is preprocessed with the distribution key TRACE on; otherwise it uses the three-argument entry points.

Table 3.1. Basic-Operation Entry Points

Operation	*Entry Point*	*Arguments*	*See*
Open device	DMOPEN	LDI, EDNAME, DDPARS, TRACE	§3.1
	DMDAST	LDI, EDNAME, DDPARS	
Close device	DMCLOS	LDI, DELETE, 0, TRACE	§3.2
	DMFAST	LDI, DELETE, 0	
Position device	DMPOST	LDI, DLOC, MODE, TRACE	§3.3
	DMPAST	LDI, DLOC, MODE	
Write device	DMWRIT	LDI, ARRAY, SIZE, TRACE	§3.4
	DMWAST	LDI, ARRAY, SIZE	
Read device	DMREAD	LDI, ARRAY, SIZE, TRACE	§3.5
	DMRAST	LDI, ARRAY, SIZE	
List information	DMSTAT	KEY	§3.6
	DMLAST	LOSD, LPKT, LTAB	

§3.1 OPEN DEVICE: DMOPEN/DMDAST

This operation opens (activates, assigns, declares) a logical device resident on main or auxiliary storage. A device must be opened before any I/O activity is attempted on it.

§3.1.1. Entry DMOPEN

The calling sequence is:

```
                    CALL DMOPEN (LDI, EDNAME, DDPARS, TRACE)
```

where

LDI	If LDI $>$ 0, index of logical device to be opened. Should this LDI be active, the old device is closed first (see Remark 3.2).
	If LDI $=$ 0 on entry, scan the Logical Device Table for an *already active* EDNAME. If found, its LDI is ***returned*** in this argument (which must therefore be a variable in the calling program), and the open operation skipped. If not found, then search the Logical Device Table for the first *inactive* LDI, set LDI to this value, and continue as in the LDI $>$ 0 case.
	If LDI $<$ 0, begin as if LDI $=$ 0, but if an active EDNAME is not found, set LDI to $\|$LDI$\|$ and then proceed as in the LDI $>$ 0 case. The absolute value is *returned* in the argument. (Note that if $\|$LDI$\|$ happens to be active on entry, the old device will be closed first.)
EDNAME	A character string containing the *external device name* described in §2.7. This text string must be supplied *left-adjusted and blank filled.* The name is assumed to be terminated by the *first occurrence of a blank character*, or by the implied length of EDNAME, whichever occurs first. The reader is referred to Sections 2.7.1-2.7.3 regarding legal device names for specific computers.
	If a blank value is specified for this argument (*i.e.*, EDNAME = ' '), a default name is selected following the rules set forth in Table 2.8.
DDPARS	A four-word integer array that supplies the *device descriptor parameters* discussed in §2.5.
	DDPARS(1) = TYPEX: device type index (see §2.5.1).
	DDPARS(2) = OPTX: device options index (see §2.5.2).
	DDPARS(3) = LIMIT: device capacity limit in words (§2.5.3) if a new or scratch device. If zero, the default size specified in Table 2.6 is assumed.

For a core-resident device (TYPEX = 5), LIMIT is the effective blank-common length allocated, starting at the offset prescribed in DDPARS(4).

DDPARS(4) = XPRU for an auxiliary storage device (TYPEX $\leq$ 4), or BCOFF for a core (blank-common-resident) device (TYPEX = 5).

For an auxiliary storage device:

XPRU > 0: external PRU size in words. Must comply with restrictions noted in §2.5.4.

XPRU = 0: select XPRU = 1 (word addressing).

XPRU = -1: select XPRU = 1 *and* buffer I/O to this device if a page buffer has been previously declared. If no buffer has been specified, XPRU = -1 is the same as 0 or 1.

For a core device, BCOFF is the blank-common offset in words of the device storage allocation. If BCOFF = 0, the device allocation is to start at the first word in blank common. For these devices, XPRU = 1 is implied.

TRACE A *positive* integer used as identifying label in error traceback. *Don't* put a zero or negative value here; these are reserved for internal use.

REMARK 3.2

The "free LDI if busy" strategy has many important applications. But sometimes it can lead to problems; if this is the case, the user program should either make use of the LDI = 0 feature, or first call LMLDIF (§5.7) to obtain a free slot in the Logical Device Table.

REMARK 3.3

If the device name of a newly created permanent file clashes with that of an existing file, an error results on systems without automated file cycling. The Univac DMGASP tries to circumvent this problem for *Block I/O devices* by cyclically changing the last character in EDNAME and resubmitting the request. Sometimes this contrivance works, sometimes it doesn't.

§3.1.2. Entry DMDAST

Entry point DMDAST performs the same service as DMOPEN, but with TRACE omitted from the calling sequence:

```
CALL DMDAST (LDI, EDNAME, DDPARS)
```

DMDAST is compatible with the old IOM versions (the DAST is for "declare auxiliary storage") and will be kept as alternate entry point indefinitely, since DMOPEN in fact calls DMDAST.

§3.1.3. Open Message

The open function writes an informative message on the bulk-print file. For an auxiliary-storage device, the format is typified by the example

```
+++ OPEN,  Ldi:  8, File:  RES.GAL, Attr:  Block I/O, NEW, Paged
```

which is largely self-explanatory. The message above is for a Paged Block I/O device, created on permanent file RES.GAL (a VAX filename) and which will be referenced through LDI number 8. For a FORTRAN I/O device, the logical unit number will be shown before the Attributes text.

The open message for a core-resident device is more concise. Example:

```
+++ OPEN,  Ldi:  12, BC( 30001:  75000)
```

This says that LDI number 12 will point to a core device that occupies word locations 30001 through 75000 of blank common. No device name is shown.

REMARK 3.4

The message is written out *just before* the open request is submitted to either the operating system or the FORTRAN I/O library. Thus, the appearance of the message does not necessarily mean that the operation was successful. If an error condition is detected, a diagnostic will immediately follow (assuming, of course, that the error-file unit is the same as the bulk-print-file unit).

REMARK 3.5

On the Univac version, the message given for *Block I/O devices* has a different format. It is the image of the @ASG request submitted to the Exec-1100 system, followed immediately by the image of the @USE request that links the external and internal file names. For FORTRAN I/O devices, the message has the standard format shown above.

REMARK 3.6

If the case of a conditional open (LDI ≤ 0), no message appears if the operation is skipped because EDNAME is already open. Otherwise the message will display the actual LDI chosen by the I/O manager.

REMARK 3.7

Some NICE programmers view these messages as nuisances, especially in highly interactive graphic processors when the bulk-print-file unit is assigned to the screen. The messages may be suppressed (forever or temporarily, as desired), by calling entry point DMSOCM (§4.7).

§3.2 CLOSE DEVICE: DMCLOS/DMFAST

This operation breaks the connection between a Logical Device Index (LDI) and the associated storage facilities. The storage resources are released to the operating system, and cease to exist if the device was of scratch type.

§3.2.1. Entry DMCLOS

The calling sequence is

```
CALL DMCLOS (LDI, DELETE, 0, TRACE)
```

where

LDI	If greater than 0, Logical Device Index of device to be closed. If this LDI is not active, no operation is performed. If zero, close *all* active devices. LDI < 0 means *conditional* close. If the "NICE macroprocessor" flag has been set on using DMACRO (§4.2), the close request is ignored. Otherwise device |LDI| is closed.
DELETE	If zero, do a normal close. If one, close and *delete* if device resides on a permanent disk file. See Remark 3.9.
TRACE	A *positive* integer used as identifying label in error traceback. *Don't* put a zero or negative value here; these are reserved for internal use.

REMARK 3.8

A nonzero third argument is used to flag calls by DMOPEN.

REMARK 3.9

Close-and-delete may not always work correctly for Block I/O devices on some operating systems. If you encounter problems, perform a normal close (DELETE = 0) and delete the file *after* the run.

REMARK 3.10

The close-all option (LDI = 0) is sometimes handy for run-termination routines. But *don't* use it if GAL or DAL files may be among the active devices (see following remark).

REMARK 3.11

Never use DMCLOS (or DMFAST) to close a GAL or DAL device under EZ-GAL control: you may leave data stranded in the header-TOC buffers! Use GMCLOS [2] instead.

REMARK 3.12

Explicit closing is important for Paged I/O devices that have been written on during the run unless periodically flushed (see Sections 2.4.3-2.4.4).

REMARK 3.13

On some systems, such as CDC's SCOPE, explicit closing of newly created permanent files or modified "old" permanent files is essential to the file survival.

§3.2.2. Entry DMFAST

Entry point DMFAST performs the same close service as DMCLOS (with an exception noted below), but with TRACE omitted from the calling sequence:

```
CALL DMFAST (LDI, DELETE, 0)
```

where the meaning of the first two arguments is the same as in DMCLOS. *The "close-all" specification LDI = 0, however, is not recognized.*

DMFAST is compatible with the old versions of the I/O Manager. FAST stands for "free auxiliary storage", which is Univac's terminology for file closing.

§3.2.3. Close Message

The close function writes an informative message on the bulk-print file. For an auxiliary-storage device, the format is typified by the example

```
+++ CLOSE, Ldi:  8, File:  RES.GAL
```

which is self-explanatory.

REMARK 3.14

As in the case of the OPEN message (§3.1.3), the CLOSE message is written out before the I/O service is requested, and an error diagnostic may follow. However, close-file errors are comparatively rare.

REMARK 3.15

On Univac, the message format for a *Block I/O device* is different: it will show the @FREE image submitted to the Exec-1100 operating system. For FORTRAN I/O devices, the message has the standard form shown above.

REMARK 3.16

In the case of a conditional close, no message appears if the operation is skipped.

REMARK 3.17

Open and close messages may be altogether suppressed by calling DMSOCM (§4.7).

§3.3 POSITION DEVICE: DMPOST/DMPAST

This operation is used to position a device to the PRU location where the next read or write is to take place. This is necessary when information is to be stored or retrieved in non-sequential manner, as discussed at length in §2.3.3.

§3.3.1. Entry DMPOST

The calling sequence is:

```
CALL DMPOST (LDI, DLOC, MODE, TRACE)
```

where

LDI	Logical Device Index of device to be positioned.
DLOC	Integer that specifies the location, in external PRUs, to which the device is to be positioned according to argument MODE.
MODE	Flags specifying positioning mode: 1: DLOC words from start of device. 0: DLOC external PRUs from start of device. -1: DLOC words from the current device location. -2: DLOC external PRUs from the current device location. If the device is *word-addressable*, MODE = 1 and 0 are equivalent, and so are MODE = -1 and -2.
TRACE	A *positive* integer used as identifying label in error traceback. *Don't* put a zero or negative value here; these are reserved for internal use.

REMARK 3.18

If the device is not word-addressable (*i.e.*, XPRU >1), and MODE is 1 or -1, the I/O Manager converts the word count given in argument DLOC to a "covering" external-PRU count as follows. Let

$$\text{COVPRU} = (\text{IABS(DLOC)+XPRU-1})/\text{XPRU}$$

where all variables are of type integer, and FORTRAN truncated-division rules are used. Then the new device location (NEWLOC) is

$$\text{NEWLOC} = \text{COVPRU} \quad \text{if MODE} = 1$$

$$\text{NEWLOC} = \text{CDLOC} + \text{ISIGN(COVPRU,DLOC)} \quad \text{if MODE} = -1$$

This is checked for validity, and inserted in the Logical Device Table overwriting CDLOC.

§3.3.2. Entry DMPAST

Entry point DMPAST performs the same service as DMOPEN, but with TRACE omitted from the calling sequence:

```
CALL DMPAST (LDI, DLOC, MODE)
```

DMPAST is compatible with the old versions of the I/O Manager.

§3.4 WRITE RECORD: DMWRIT/DMWAST

The write operation transfers one IOM record from a specified location in main storage to a logical device starting at the current device location.

§3.4.1. Entry DMWRIT

The calling sequence is:

```
CALL DMWRIT (LDI, ARRAY, SIZE, TRACE)
```

where

LDI	Logical Device Index.
ARRAY	A *numeric* array that contains the record to be transmitted. Avoid use of character variables; see Remarks 3.17 and 3.18.
SIZE	Size of records in words; must be greater than zero.
TRACE	A *positive* integer used as identifying label in error traceback. *Don't* use a zero or negative value; these are reserved for internal use.

The record is stored beginning at the current device location (CDLOC) of device LDI. On an error-free return from DMWAST, the current device location is advanced by

$$1 + (\text{SIZE} - 1)/\text{XPRU}$$

where XPRU (integer) is the external PRU size. If the device is word-addressable, XPRU = 1, and the location is simply advanced by SIZE.

If an error condition occurs, the device location is *not* updated.

REMARK 3.19

The formula for updating CDLOC is the same regardless of whether the device is Paged or Unpaged. For Paged devices, the location update is strictly logical and generally bears no relation to data transfers between the Page Buffer Pool and disk files (physical writes to *other* Paged devices may result as side effect of page exchanges).

REMARK 3.20

If the current device location (CDLOC) happens to lie beyond the end-of-information, the CDC version of DMGASP extends the EOI with zero-filled dummy records in the case of Block I/O devices.

REMARK 3.21

If ARRAY is of type CHARACTER, proper handling of subroutine linkage in VAX-FORTRAN requires an explicit call-by-reference. This means that the second argument must be specified as

%REF(ARRAY)

Even with this modification, it should be remembered that DMWRIT writes only *full words* (4-character groups on the VAX or similar 32-bit machines). Thus, byte-addressing has to be simulated at higher levels than DMGASP.

REMARK 3.22

Univac and CDC FORTRAN compilers will accept an ARRAY argument of type CHARACTER without complaining. The result of the write operation will be generally *wrong*, however, unless ARRAY happens to be *word-aligned*, because neither DMWRIT nor DMWAST account for character offsets in these word-addressable machines. Safe handling of a character record require prior copy to a "scratch" Hollerith array, which is then presented to DMWRIT or DMWAST.

§3.4.2. Entry DMWAST

Alternate entry point DMWAST provides the same write-record service as DMWRIT, but lacks the TRACE argument:

```
CALL DMWAST (LDI, ARRAY, SIZE)
```

This entry point is compatible with previous versions of the I/O Manager.

§3.5 RECORD: DMREAD/DMRAST

The read operation transfers one IOM record from a logical device, starting at the current device location, to a specified location in main storage.

§3.5.1. Entry DMREAD

The calling sequence is:

```
CALL DMREAD (LDI, ARRAY, SIZE, TRACE)
```

where

LDI	Logical Device Index.
ARRAY	A numeric array that will receive the record.
SIZE	Size of record in words; must be greater than zero.
TRACE	A *positive* integer used as identifying label in error traceback. *Don't* use a zero or negative value; these are reserved for internal use.

The record is read starting at the current device location (CDLOC) of device LDI. On an error-free return from DMREAD, the current device location is advanced by

$$1 + (\mathrm{SIZE} - 1)/\mathrm{XPRU}$$

where XPRU (integer) is the external PRU size. If the device is word-addressable, XPRU = 1, and the location is simply advanced by SIZE.

If an error condition is detected, the device location is *not* updated.

REMARK 3.23

The formula for updating CDLOC is the same regardless of whether the device is Paged or Unpaged. For Paged devices, the update is strictly logical and generally bears no resemblance to actual data transfers between the Page Buffer Pool and the disk (physical writes to *other* Paged devices may occur as a side result of page exchanges).

REMARK 3.24

If the resulting new device location would overshoot the end-of-information, an error condition is diagnosed and no read occurs.

REMARK 3.25

If ARRAY is of type CHARACTER, proper handling of subroutine linkage under VAX-FORTRAN requires an explicit call-by-reference. This means that the second argument must be specified as

%REF(ARRAY)

Even with this modification, it should be remembered that DMREAD moves only ***full words*** (4-character groups on the VAX and similar 32-bit machines). Thus, byte-addressing has to be simulated at higher levels than DMGASP.

REMARK 3.26

Univac and CDC FORTRAN compilers will accept an ARRAY argument of type CHARACTER without complaining. The result of the DMREAD operation will be generally ***wrong***, however, unless ARRAY happens to be ***word-aligned***, because neither DMREAD nor DMRAST account for character offset on these word-addressable machines. Safe handling of a character record requires a read into a "scratch" Hollerith array, which is then copied to the destination character string.

§3.5.2. **Entry DMRAST**

Alternate entry point DMRAST provides the same read-record service as DMREAD, but lacks the TRACE argument:

```
CALL DMRAST (LDI, ARRAY, SIZE)
```

This entry point is compatible with previous versions of the I/O Manager.

§3.6 LIST INFORMATION: DMSTAT/DMLAST

Entry point DMSTAT or DMLAST may be accessed to print selected state information related to the activities of the I/O manager.

§3.6.1. Entry DMSTAT

The calling sequence is:

```
CALL DMSTAT (KEY)
```

where KEY is a four-character string that specifies the information to be printed:

KEY = 'LDT '	Print Logical Device Table, showing active devices only. Print legends are explained in Table 3.2.
KEY = 'LDTF'	Print Logical Device Table in full.
KEY = 'PBT '	Print contents of Page Buffer Table.
KEY = 'PIOS'	Print Paged I/O statistics if this feature has been used. Interpretation of these data requires familiarity with paging systems, however.
KEY = 'PKT '	Print I/O Packet used to set up Block I/O requests. Interpretation of these data requires system-level expertise.
KEY = 'OSD '	Print Operation Status Descriptors. This is an array of variables which retains information about the last traceable IOM operation performed. OSD display is primarily useful after error conditions. Print legends are explained in Table 3.3.

All output produced by DMSTAT goes to the bulk-print file, which is by default unit 6. (NICE programmers may reset this unit number through CLIP's *PRT directive.)

Table 3.2
LDT Print Explanation

Print Caption	*Explanation*
Ldi	Logical Device Index
External DevName	External Device Name as stored in the LDT (may continue on the next print line)
Unit	FORTRAN Logical Unit paired with this LDI; meaningless if Block I/O or core device
EC	Equipment Code: a nonzero value flags an active device; negative value flags core device
Typ	Codified device type: BIO for Block I/O, FDA for FORTRAN Direct-Access, BC for blank-common resident
Sta	Codified device status: NEW, OLD, SCRatch as determined by the device options index
Ext PRU	External PRU in words
Int PRU	Internal PRU in words
Cdloc	Current device location in external PRUs
Next	Next free location in external PRUs
Limit	Device capacity limit in external PRUs
Userwords read	Counter of words retrieved by the *user program*; computed as sum of sizes of records read successfully to this device using DMREAD/DMRAST since the device was opened. (For some device types, this value may differ significantly from a physical-word-read count)

Table 3.2
LDT Print Explanation (concluded)

Print Caption	*Explanation*
Userwords written	Counter of words stored by the *user program*; computed as sum of sizes of records written successfully to this device using DMWRIT/DMWAST since the device was opened. (Same comment as above)
Active devices	Self-explanatory
Full devices	Count of devices for which a "device capacity exceeded" error has occurred
Reads	Counts of calls made to DMREAD/DMRAST for *all* devices since run start
Writes	Counts of calls made to DMWRIT/DMWAST for *all* devices since run start
Words Xfd	Two values follow. The first is the sum of userwords transferred using successful calls to DMREAD/DMRAST and DMWRIT/DMWAST for *all* devices since run start. The second value is similarly defined, but measures *physical* words transferred.

Table 3.3
OSD Print Explanation

Print Caption	*Explanation*
Last TRACE-able Entry	Shows which of the following IOM entry points was called last: DMOPEN/DMDAST, DMCLOS/DMFAST, DMFLUB, DMPOST/DMPAST DMREAD/DMRAST, DMWRIT/DMWAST. Following values are "leftovers" from this entry point.
Ioercd	I/O error code (cf. §6.1) on exit from last traceable entry
Ldi	LDI value supplied as 1st argument to last entry
Typex	Device type index for the LDI shown (DDPARS supplied if last entry was DMOPEN/DMDAST)
Optx	Device option index for the LDI shown (DDPARS supplied if last entry was DMOPEN/DMDAST)
Lcarg1,Lcarg2	DLOC, MODE arguments of DMPOST/DMPAST if this was the last entry, otherwise zero
Locdev	Current device location in external PRUs
Sizrec	Record size given to DMWRIT/DMWAST or DMREAD/DMRAST if last entry; otherwise zero
Nwxwrd	"User words" actually transferred by DMWRIT/ DMWAST or DMREAD/DMRAST if last entry. May be less than Sizrec if an abnormal condition was detected.
I/O status	The I/O status value (cf. §6.1) returned by either the O/S (Block I/O) or FRTL (FORTRAN I/O) on the last service request. Nonzero O/S values are not necessarily error codes on all systems. Some IOM versions show *two values here.*

§3.6.2. Entry DMLAST

Entry point DMLAST provides some of the print services of DMSTAT. It is provided for compatibility with earlier versions of the I/O manager.

The calling sequence is:

```
CALL DMLAST (LOSD, LPKT, LTAB)
```

where

LOSD	If nonzero, print Operation Status Descriptors.
LPKT	If nonzero, calls for a listing of the first (LPKT+1) words of the I/O Packet array for Block I/O.
LTAB	If greater than zero, calls for a listing of the Logical Device Table showing only active devices. If LTAB is zero, the LDT is not printed. If LTAB = -1, the *complete* LDT is printed, including inactive devices.

THIS PAGE LEFT BLANK INTENTIONALLY.

4 Supplemental Operations

Section 4: SUPPLEMENTAL OPERATIONS

In addition to the basic services described in Section 3, DMGASP provides a set of supplemental entry points to perform advanced or specialized operations.

These entry points are alphabetically listed in Table 4.1, and covered in Sections 4.1-4.9. The user should be warned that, with the exception of DMFLUB, none of these entry points check for the validity of its inputs.

Table 4.1. Supplemental-Operation Entry Points

Operation	*Entry Point*	*Arguments*	*See*
Error-abort run	DMABRT		§4.1
Set macroprocessor flag	DMACRO	MF	§4.2
Flush Page Buffer Pool	DMFLUB	0, 0, TRACE	§4.3
Set device capacity limit	DMLIMT	LDI, LIMIT	§4.4
Set device extent	DMNEXT	LDI, NEXT	§4.5
Declare Page Buffer Pool	DMPOOL	PB, LP, NP	§4.6
Suppress open-close messages	DMSOCM	M	§4.7
Reset LDI-to-unit table	DMUNIT	LDI, NLDI, UNIT	§4.8
Set external PRU size	DMXPRU	LDI, XPRU	§4.9

§4.1 ERROR-ABORT RUN: DMABRT

Entry point DMABRT generates an abnormal run stop.

The calling sequence is:

```
CALL DMABRT
```

On most computers, the error-abort is forced using a division by zero.

§4.2 SET MACROPROCESSOR FLAG: DMACRO

DMACRO can be used to turn the "NICE macroprocessor" flag on or off. Setting this flag to on affects the outcome of conditional-close operations (§3.2).

The calling sequence is:

```
CALL DMACRO (MP)
```

Setting MP = 1 turns the macroprocessor flag on, while MP = 0 turns it off. The default state is zero.

§4.3 FLUSH PAGE BUFFER POOL: DMFLUB

DMFLUB scans the Page Buffer Pool (PBP) for modified pages belonging to non-scratch devices, and writes them out. This operation protects data integrity on permanent files in the event of an abnormal run termination (for explanation, see §2.4).

The calling sequence is:

```
CALL DMFLUB (0, 0, TRACE)
```

where

TRACE	A *positive* integer used as identifying label in error traceback prints.

REMARK 4.1

Nonzero first and second arguments are reserved for internal use.

REMARK 4.2

If no Page Buffer Pool has ever been defined, or no Paged I/O devices exist, DMFLUB does nothing.

§4.4 SET DEVICE CAPACITY LIMIT: DMLIMT

Entry point DMLIMT can be used to reset the device capacity limit (LIMIT) on a device to a specified value. This overrides the value supplied in DDPARS at device-open time (§3.1).

The calling sequence is:

```
CALL DMLIMT (LDI, LIMIT)
```

where

LDI	Logical Device Index.
LIMIT	The value of LIMIT in *external PRUs.*

DMLIMT has applications similar to that of DMNEXT (§4.5), but is rarely used.

REMARK 4.3

DMLIMT does not check for the validity of its arguments.

REMARK 4.4

Note that LIMIT is given in external PRU units and not in words, as was the case for DDPARS(3) in §3.1.

REMARK 4.5

DMLIMT has effect only on *active* devices. Using DMLIMT on an inactive device is meaningless but causes no harm.

§4.5 SET DEVICE EXTENT: DMNEXT

Entry point DMNEXT can be used to set the end of information (NEXT) on a device to a specified value. This value characterizes the device extent.

The calling sequence is:

```
CALL DMNEXT (LDI, NEXT)
```

where

LDI	Logical Device Index.
NEXT	The value of NEXT in *external PRUs.*

The main application of DMNEXT is to set the exact length of a *just-open* logical device resident on a permanent disk file when this length is maintained in the file itself. Motivation for this is given in §2.5.6. The procedure is illustrated by the following example.

A new permanent, word-addressable file is opened and written. When the user program is through with the file, its length in words (which may be retrieved from LMNEXT, §5.10), is stored in a reserved "header record" at the file start. The file is closed.

On a subsequent run, the file is reopened and the header record read. The exact length is retrieved, and supplied to the I/O Manager through DMNEXT.

REMARK 4.6

DMNEXT does not check for the validity of its arguments.

REMARK 4.7

DMNEXT has effect only on *active* devices. Using DMNEXT on an inactive device is meaningless though it causes no harm.

REMARK 4.8

If DMNEXT is not used, the I/O Manager sets an approximate (and conservative) value for NEXT obtained from system information. But if it can't get any information (as in the case of FORTRAN I/O devices), it sets NEXT = LIMIT.

§4.6 DECLARE PAGE BUFFER POOL: DMPOOL

Entry point DMPOOL declares a Page Buffer Pool (PBP) for subsequent use in Paged I/O support.

The calling sequence is:

```
                    CALL DMPOOL (PB, LP, NP)
```

where

PB	An integer array dimensioned LP*NP + 2*NP + 2 words which will be used by the I/O manager as workspace for Page Buffer Pool (LP*NP words) and Page Buffer Table (2*NP words). Two words are used to store protection data.
LP	Page length in words. Must be an *exact multiple* of the internal PRU size (§2.2.4 and Table 2.2) for optimal I/O efficiency. Best results are generally achieved when LP is 4 to 16 times the internal PRU (see Appendix C). If LP $\leq$ 0, the PBP declaration is ignored, and no diagnostics are given.
NP	The number of pages in the buffer. As a very rough guide, NP should be of the order of 10 times the number of Paged I/O devices that may be simultaneously active. If NP $\leq$ 0, the PBP declaration is ignored, and no diagnostics are given.

REMARK 4.9

DMPOOL must be called before any Paged I/O device is opened. A good place to put the call is at the start of the user program.

REMARK 4.10

Assuming that LP is greater than 0 and NP is greater than 0, DMPOOL performs the following actions: saves LP and NP, computes and saves the blank-common address of PB (which, however, doesn't have to be in blank common), clears the workspace, and stores protection keys.

REMARK 4.11

Once the PB array is specified using DMPOOL, the user program should never modify it.

REMARK 4.12

All Paged I/O devices subsequently opened will *share* the Buffer Pool. The number of Paged I/O devices that are simultaneously active should be no more than NP/8 to prevent thrashing.

REMARK 4.13

If Paged I/O devices have different internal PRU sizes (a rare event), try to make LP a common multiple of the internal PRU sizes.

REMARK 4.14

DMPOOL could conceivably be called more than once during a run with the same or different arguments. But be sure that ***all Paged I/O devices are closed*** before calling it anew.

§4.7 SUPPRESS OPEN/CLOSE MESSAGES: DMSOCM

Entry point DMSOCM may be used to suppress permanently or temporarily informative messages printed by the I/O manager when opening and closing logical devices (Sections 3.1.3, 3.2.3).

The calling sequence is:

```
CALL DMSOCM (M)
```

where M is the number of subsequent messages to be suppressed.

M	If M > 0, suppress the next M messages. For permanent suppression, make M large, *e.g.*, M = 10000.
	If M = 0, print is restored.

§4.8 RESET LDI-UNIT TABLE: DMUNIT

Entry point DMUNIT may be used to reset the LDI-to-logical-unit correspondence table.

The calling sequence is:

```
CALL DMUNIT (LDI, NLDI, UNIT)
```

where

LDI	Logical Device Index of first device whose logical unit is to be changed.
NLDI	Number of logical units to be changed; these correspond to devices LDI, ..., LDI+NLDI-1.
UNIT	Logical unit for device LDI (integer). Logical units for devices LDI+1, ..., LDI+NLDI-1 are generated by incrementing UNIT.

An example should clarify the generation scheme. Suppose that the user-program developer wishes to use

11, 12, ..., 18

as logical units associated to logical devices 1 through 8. The following call would accomplish that:

```
CALL DMUNIT (1, 8, 11)
```

REMARK 4.15

DMUNIT does not check for the validity of its arguments.

REMARK 4.16

Never change the logical unit of an *active* LDI. In fact, DMUNIT should be called at the start of the program, before any device activity occurs.

§4.9 SET EXTERNAL PRU SIZE: DMXPRU

Entry point DMXPRU may be used to reset the external PRU size of a device to a specified value.

The calling sequence is:

```
CALL DMXPRU (LDI, XPRU)
```

where

LDI	Logical Device Index.
XPRU	External PRU size in words.

This has applications similar to those discussed for DMNEXT (§4.5), but is more exotic and daring.

REMARK 4.17

DMXPRU does not check for the validity of its arguments.

REMARK 4.18

DMXPRU has effect only on *active* devices. Using DMXPRU on an inactive device is meaningless but causes no harm.

THIS PAGE LEFT BLANK INTENTIONALLY.

5 Information Retrieval Functions

The I/O Manager provides a comprehensive set of entry points that return state information maintained in its internal tables. These are referenced as integer functions of the form LM*xxxx*, where *xxxx* is a mnemonic identifier.

Table 5.1 lists alphabetically the information-retrieval entry points discussed in Sections 5.1-5.15.

REMARK 5.1

Information retrieval functions pertaining to error-handling (*e.g.*, LMERCD, LMIOST) are covered in Section 6.

REMARK 5.2

None of these functions check for legal input arguments.

REMARK 5.3

In early DMGASP versions, this information could be directly extracted by the user program from a named common block. Experience has shown, however, that this practice had detrimental effects on program modularity and transportability. Hence the provision of a comprehensive set of information-retrieval functions has been implemented.

Table 5.1. Information-Retrieval Functions

Operation	*Entry Point*	*Arguments*	*See*
Retrieve device location	LMDLOC	LDI	§5.1
Retrieve equipment code	LMEQCD	LDI	§5.2
Inquire by Logical Device	LMINQL	LDI, EDN, DDP	§5.3
Inquire for name in LDT	LMINQT	EDN	§5.4
Inquire for file existence	LMINQX	EDN	§5.5
Retrieve internal PRU	LMIPRU	LDI	§5.6
Retrieve first free LDI	LMLDIF	LDIBEG	§5.5
Retrieve LDI that matches unit	LMLDIU	UNIT	§5.8
Retrieve device capacity limit	LMLIMT	LDI	§5.9
Retrieve device extent	LMNEXT	LDI	§5.10
Retrieve userwords written	LMNUWW	LDI	§5.11
Retrieve options index	LMOPTX	LDI	§5.12
Retrieve type index	LMTYPX	LDI	§5.13
Retrieve logical unit	LMUNIT	LDI	§5.14
Retrieve external PRU	LMXPRU	LDI	§5.15

§5.1 RETRIEVE DEVICE LOCATION: LMDLOC

Entry point LMDLOC, referenced as an integer function, returns the current location of a device identified by its LDI.

The function reference is:

```
DLOC = LMDLOC (LDI)
```

where

LDI	Logical Device Index.
LMDLOC	Returns the current device location in external PRUs. If the device is inactive, or the argument is out of range, the value returned is meaningless.

§5.2 RETRIEVE EQUIPMENT CODE: LMEQCD

Entry point LMEQCD, referenced as an integer function, returns the equipment code of a logical device identified by its LDI.

The function reference is:

```
ICODE = LMEQCD (LDI)
```

where

LDI	Logical Device Index.
LMEQCD	Returns the device equipment code: zero if device is inactive; > 0 if active and resident on auxiliary storage; -1 if active and resident in blank common (core device). If the argument is out of range, the value returned is meaningless.

REMARK 5.4

Inasmuch as equipment code numbers for auxiliary storage are machine dependent, the main use of LMEQCD is for testing whether a specific LDI is active or inactive.

§5.3 INQUIRE BY LDI: LMINQL

Entry point LMINQL inquires about the characteristics of a specific device identified by its LDI.

The function reference is:

```
ISTAT = LMINQL (LDI, EDN, DDP)
```

where the input is:

LDI	Logical Device Index.

and the outputs are:

LMINQL	Zero if device is inactive. 1 if device is active.
EDN	A character array that receives the external device name if the device is active. If the device is inactive, the character array contains blanks.
DDP	A four-word integer array. If device is active, the four device descriptors discussed in §3.1 are returned here. If the device is inactive, all four words are set to zero.

§5.4 INQUIRE FOR EDN IN LDT: LMINQT

Entry point LMINQT finds out whether its device-name argument is stored in the Logical Device Table, and if so, whether it is active or inactive.

The function reference is:

```
LDI = LMINQT (EDN)
```

where

EDN	External Device Name.
LMINQT	Zero if device name is not in LDT. +LDI if name matches EDN of LDI-th device and is active. −LDI if name matches EDN of LDI-th device but is inactive.

REMARK 5.5

On the VAX, both argument and LDT device names are expanded to full 64-character system names (using the $PARSE RMS service) before the names are compared for equality.

§5.5 INQUIRE FOR EDN EXISTENCE: LMINQX

Entry point LMINQX, referenced as an integer function, inquires about the existence of an external device name as the identifier of an existing *permanent* file.

The function reference is:

```
ISTAT = LMINQX (EDN)
```

EDN	External device name, as in §3.1.
LMINQX	0: file does not exist.
	1: file exists and is neither GAL nor DAL.
	10: file exists and is a GAL.
	11: file exists and is a DAL.
	−1: file exists but is locked by this process or another process (VAX).
	−2: file exists but is inaccessible because owner denies access (VAX).

§5.6 RETRIEVE INTERNAL PRU: LMIPRU

Entry point LMIPRU, referenced as an integer function, returns the internal physical record unit (PRU) of a device identified by its LDI.

The function reference is:

```
IPRU = LMIPRU (LDI)
```

where

LDI	Logical Device Index.
LMIPRU	Internal PRU size of device. If the device is inactive, or the argument is out of range, the value returned is meaningless.

§5.7 RETRIEVE FIRST FREE LDI: LMLDIF

Entry point LMLDIF, referenced as an integer function, returns the first free "LDI slot" in the Logical Device Table. The LDT search extent and direction is controlled by the index argument.

The function reference is:

```
LDI = LMLDIF (LDIBEG)
```

where

LDIBEG	Specifies LDT search extent and direction as follows. > 0: scan from LDIBEG (inclusive) forward. < 0: scan from highest LDI (presently 16) backwards. 0: return the highest legal LDI.
LMLDIF	If LDIBEG is nonzero, returns the first inactive LDI found. If none is found, zero is returned. If LDIBEG = 0, LDLDIF returns the highest legal LDI (presently 16).

§5.8 RETRIEVE LDI THAT MATCHES UNIT: LMLDIU

Entry point LMLDIU, referenced as an integer function, answers the question: which LDI has an associated logical unit that matches its argument?

The function reference is:

```
LDI = LMLDIU (UNIT)
```

where

UNIT	Logical unit number (a positive integer).
LMLDIU	LDI whose associated logical unit matches the argument, otherwise zero.

§5.9 RETRIEVE DEVICE CAPACITY LIMIT: LMLIMT

Entry point LMLIMT, referenced as an integer function, returns the capacity limit of a device identified by its LDI.

The function reference is:

```
LIMIT = LMLIMT (LDI)
```

where

LDI	Logical Device Index.
LMLIMT	Device capacity limit in external PRUs. If the device is inactive, or the argument is out of range, the value returned is meaningless.

REMARK 5.6

A negative LIMIT flags a "full" direct access device; *i.e.*, one that was the target of a previous write-attempt beyond its capacity limit.

§5.10 RETRIEVE DEVICE EXTENT: LMNEXT

Entry point LMNEXT, referenced as an integer function, returns the next-free-location of a device identified by its LDI. This value defines the device extent (§2.2).

The function reference is:

```
NEXT = LMNEXT (LDI)
```

where

LDI	Logical Device Index.
LMNEXT	The next-free-location in external PRUs. If the device is inactive, or the argument is out of range, the value returned is meaningless.

§5.11 RETRIEVE NUMBER OF USER WORDS WRITTEN: LMNUWW

Entry point LMNUWW, referenced as in integer function, returns the number of words written on a specific logical device since it was opened.

The function reference is:

```
NW = LMNUWW (LDI)
```

where

LDI	Logical Device Index.
LMNUWW	Number of user words written since the device was open. (The sum of sizes of records *successfully* written to this device through DMWRIT or DMWAST.) If the argument is out of range, the value returned is meaningless.

REMARK 5.7

This entry point is useful when the user program has to take some action according whether a device has been written on during the run.

REMARK 5.8

If device index LDI has been opened more than once during a run, LMNUWW returns only the number of user words transferred since the last open.

§5.12 RETRIEVE DEVICE OPTIONS INDEX: LMOPTX

Entry point LMOPTX, referenced as in integer function, returns the LDT-stored options index of a device identified by its LDI.

The function reference is:

```
OPTX = LMOPTX (LDI)
```

where

LDI	Logical Device Index.
LMOPTX	The options index presently stored in the LDT. If the device is inactive, or the argument is out of range, the value returned is meaningless.

REMARK 5.9

The value returned is not necessarily the one supplied in DDPARS(2) to DMOPEN to DMDAST (§3.1), if this was a negative value. For example, if DDPARS(2) = -6, LMOPTX returns 6 or 4, depending on whether a new device was created or not.

§5.13 RETRIEVE DEVICE TYPE INDEX: LMTYPX

Entry point LMTYPX, referenced as in integer function, returns the LDT-stored type index of a device identified by its LDI.

The function reference is:

```
TYPX = LMTYPX (LDI)
```

where

LDI	Logical Device Index.
LMTYPX	The type index presently stored in the LDT.
	If the device is inactive, or the argument is out of range, the value returned is meaningless.

§5.14 RETRIEVE LOGICAL UNIT: LMUNIT

Entry point LMUNIT, referenced as an integer function, returns the logical unit number associated with a device specified by its LDI.

The function reference is:

```
LU = LMUNIT (LDI)
```

where

LDI	Logical Device Index.
LMUNIT	Logical unit number. If the device is inactive, or the argument is out of range, the value returned is meaningless.

REMARK 5.10

This entry point may be viewed as the "dual" of LMLDIU (§5.8) because

```
LDI = LMLDIU (LMUNIT (LDI))
```

§5.15 RETRIEVE EXTERNAL PRU: LMXPRU

Entry point LMXPRU, referenced as an integer function, returns the external physical record unit (PRU) of a device identified by its LDI.

The function reference is:

```
IPRU = LMXPRU (LDI)
```

where

LDI	Logical Device Index.
LMXPRU	External PRU size of device in words. If the device is inactive, or the argument is out of range, the value returned is meaningless.

6
Error Handling

Operations described in Sections 3-4 and which contain a TRACE argument may be aborted or only partially executed on account of error conditions detected within the I/O Manager proper, or by the operating system.

This section covers error processing, explains error messages, and describes entry points that NICE programmers may use to store and retrieve error-related information, and to modify default error handling. These entry points are listed in Table 6.1.

REMARK 6.1

The term *error*, used in the present context, means lack of success in performing an action. A more accurate word would be failure. However, we shall conform here to the common term, error, because that usage is universally accepted and because failure has an extreme connotation.

C-2

Table 6.1 Error-Handling Entry Points

Operation	*Entry Point*	*Arguments*	*See*
Identify user subprogram	DMUSER	SUBNAM	§6.3
Test error condition	LMERCD	IERR	§6.4
Extract error information	DMEINF	IERR, EMSG, K	§6.5
Retrieve I/O status	LMIOST	J	§6.6
Defuse fatal errors	DMEASY	KERR	§6.7
Specify error terminator	DMETER	UPGERR	§6.8
Take fatal error exit	DMFATE	NAME, EKEY	§6.9
Print error trace stack	DMPETS	-2, PRTFIL	§6.10

§6.1 ERROR PROCESSING OVERVIEW

§6.1.1. Error Classification

The I/O Manager finds out about error conditions in two ways: either through an internal validity check, or by receiving an error indication from the I/O system. Whatever the source, the IOM calls the *central error management subroutine* DMSERR, which serves the whole of NICE-DMS.

DMSERR first logs a short message on the error print file (normally unit 6). These error messages are listed and explained in §6.2.

Next, errors are classified into three types:

1. *Warning-only.* Control returns to the calling program, and execution continues. The user program may at this point interrogate the error condition, and take appropriate action.

2. *Fatal.* Execution is terminated after more detailed printout. If the user program has specified an error-termination routine, DMSERR calls it. (Error termination routines are useful for cleanup operations such as buffer-flushing and file closing.)

3. *Catastrophic.* The run is aborted immediately. Even if an error-termination routine has been specified, it is not called.

Catastrophic errors are those that may reflect serious problems in the user program logic. For example: destruction of the Logical Device Table or the Page Buffer Pool caused by array overspill in the user program. For obvious reasons, this error type is not controllable by the user program or affected by the run environment.

Classification of non-catastrophic errors into fatal and warning-only depends on two factors: the *run environment*, and *user program specifications.* If the user program has specified nothing, the IOM uses the run environment as the only criterion:

Interactive Run. A non-catastrophic error is treated as warning-only, unless a total error count maintained by DMSERR exceeds an internally set limit (usually 50). If the error count limit is exceeded, a fatal error exit is taken.

Batch Run. A non-catastrophic error is treated as fatal.

How does the IOM know about the run environment? On first entry, it queries the operating system for such information, and saves the answer in its internal tables.

The preceding "default" treatment can be modified, within certain limits, by the user program through entry points DMEASY (§6.7) and DMETER (§6.8).

§6.1.2. Error Terminology

Applications programmers making use of NICE-DMS should be aware of the following terminology, which is used in subsequent sections.

Error code	An integer value which is set to a nonzero value when an error condition occurs.
Error key	A four-letter character string that uniquely specifies the error type.
Error message	The diagnostic text placed by DMSERR on the error print file.
Error trace stack	The ETS is a data structure optionally maintained by NICE-DMS, and which records the tree of internal calls. (The presence or absence of ETS depends on parameterization of the EZ-GAL and DM-GASP master-source-code preprocessing prior to compilation.)
I/O status	An integer value, or set of integer values, returned by the operating system to identify errors detected in an I/O transaction. The IOM saves this value (or values) in an internal array.

§6.2 ERROR DIAGNOSTICS

§6.2.1. Error Message Format

Error messages issued by DMSERR are of the form

****Subnam** EKEY, diagnostic text**

where *Subnam* is the name of the subroutine that calls DMSERR (often the same subroutine that detected the error), EKEY is a four-letter error key, and "diagnostic text" is a short explanatory message.

This message may be followed by one or two additional lines that furnish additional details such as the I/O status value.

Note the disappearance of *error code numbers* from the message. In the present DMGASP version, error codes have less importance than in previous versions.

§6.2.2. List of Error Messages

All possible DMSERR error messages are listed below in key-alphabetical order. Many of these are native to the global database manager EZ-GAL and are included here for completeness only.

In the following message list, items in *italics* denote variable names or numbers that are printed as part of the error message.

CFDS, Cannot find dataset

EZ-GAL level error.

CRTB, Char record too big

EZ-GAL level error.

DCLE, Device close error, file: *Filename*

The operating system has reported an error during a device-close operation. This is a very unusual condition. Track the I/O status code for further details.

DCOE, Device connect error, file: *Filename*

This can only occur for VAX/VMS Block I/O devices. The VAX/VMS record management service (RMS) has reported an error condition when trying to carry out a file-connect service.

DEXE, Device extend error, file: ***Filename***

Not presently active; reserved for future implementations.

DINE, Device inquire error, file: ***Filename***

A device-existence query performed through a FORTRAN INQUIRE statement caused an error return.

DIRO, Device is read-only

A write-record operation was attempted on a device opened in read-only mode. The operation is ignored.

DNCL, Device not connected to library

EZ-GAL level error.

DNDA, Device is not direct access

EZ-GAL level error.

DNWA, Device is not word addressable

EZ-GAL level error.

DOPE, Device open error, file: ***Filename***

A device-open operation failed. This is a common error, especially in interactive work. If declaring an existing (OLD) file, the most likely causes are:

1. Illegal file name.
2. File does not exist.
3. File has write-permission denied (write-locked) by the user program, or another program.
4. Access permission denied by file owner.

If file is created by the open operation (NEW or SCRATCH):

1. Illegal file name.
2. On some operating systems such as CDC's NOS: filename duplicates that of an existing catalogued file.
3. On VAX: file creation was attempted on a directory that denies write permission.

If the error cause is not evident, look up the status code printed on the next line in the appropriate system manual.

DOVF, Device overflow

A write-record operation would have exceeded the device capacity limit. The operation is aborted.

DQEX, Disk quota exceeded

A write-record operation is aborted by the operating system as the disk quota would be exceeded (VAX).

DSCX, Dataset capacity exceeded

EZ-GAL level error (catastrophic).

DSNT, Dataset is not text

EZ-GAL level error.

DTNA, Device type not available

A device type is not available on the IOM version being used. Most common: trying a TYPEX such as 2 (Univac drum), which no longer exists.

DTNC, Device type not creation's

EZ-GAL level error.

FACD, File already connected to other LDI

A device opened with OLD status is already active on another LDI. The device-open is aborted. Declaring on the *same* LDI is permitted, however, as the previous device is automatically closed. Declaring NEW or SCRATCH is also permitted on computers like the VAX, as the system simply increments the file cycle or version number.

FNGD, File not GAL or DAL

EZ-GAL level error.

ILDP, Illegal device position

The result of a positioning operation using DMPOST or DMPAST would result in the new device position being either negative or over the device capacity limit. The new position is not stored.

ILDI, Illegal LDI

A Logical Device Index (LDI) is outside the legal range 1 through 16. A very common error in interactive works.

ILDS, Illegal dataset name

EZ-GAL level error.

ILOI, Illegal OPTX index

The device-assignment options index (OPTX) supplied to either DMOPEN or DMDAST is outside the legal range -6 to 12. The device-open operation is aborted.

ILOP, Illegal operation

EZ-GAL level error.

ILRS, Illegal record size

The size of a record presented to a record-transfer entry point is zero or negative.

ILSN, Illegal sequence number

EZ-GAL level error.

ILTI, Illegal type index

The device type index (TYPEX) presented to DMOPEN or DMDAST is outside the legal range -4 to 5.

ILXP, Illegal external PRU

An external PRU size presented to DMOPEN or DMDAST does not exactly divide the internal PRU size (example: internal PRU 128 words, external PRU 24 words). This can never happen if the word-addressable default is used, which is the recommended setting.

INDI, Inactive LDI

An I/O operation is attempted on a device that has not been previously opened.

LDTD, Logical Device Table destroyed

A protection key stored in front of the auxiliary storage tables has been destroyed. This is considered a catastrophic error.

LDTF, Logical Device Table full

Open-device request refused because all 16 slots in the Logical Device Table are in use.

MIRE, Miscellaneous read error
MIWE, Miscellaneous write error

These are "catch-all" errors for data-transfer situations that cannot be easily categorized. If the cause is not immediately apparent, and usually is not, the recommended procedure is to record the I/O status code printed on the next line, and refer to the appropriate system manual.

MROL, Modification of read-only library ignored

EZ-GAL level error.

NRFD, No room for descriptor

EZ-GAL level error.

ODDS, Operation on deleted dataset

EZ-GAL level error.

PBPD, Page Buffer Pool destroyed

A protection key stored in front of the Page Buffer Pool has been altered. This is considered a catastrophic error.

RBEI, Read beyond end of information

A read operation through DMREAD or DMRAST specifies a record that extends beyond the end of information (NEXT). The operation is ignored.

RBTS, Record buffer too small

EZ-GAL level error.

RODS, Read outside dataset ***Dsname***

EZ-GAL error.

SONA, Sequential operation not available

An operation other than open or close has been specified on a sequential-access device, *i.e.*, one opened with a negative TYPEX.

TMOL, Too many open libraries

EZ-GAL error.

WODS, Write outside dataset ***Dsname***

EZ-GAL error.

§6.3 IDENTIFY USER SUBPROGRAM: DMUSER

The first executable statement of any user-program subroutine that calls a TRACE-equipped entry point should be a call to DMUSER.

The calling sequence is:

```
CALL DMUSER (SUBNAM)
```

where

SUBNAM	A character string of up to eight characters that identifies the user-program subroutine (normally the subroutine name).

REMARK 6.2

This name will appear at the "base" of ETS (Error Trace Stack) printouts.

REMARK 6.3

At the EZ-GAL level, this entry point is known as GMUSER, which has the identical effect and the same calling sequence.

REMARK 6.4

Before any call to DMUSER (or GMUSER) is made, NICE-DMS assumes 'USRPRG' as its ETS-base identifier.

REMARK 6.5

Programs that access the I/O manager level only (not EZ-GAL) and only reference the old TRACE-less entry points (*e.g.*, DMDAST in lieu of DMOPEN, and so forth) need not call DMUSER.

§6.4 TEST ERROR CONDITION: LMERCD

Entry point LMERCD, referenced as an integer function, furnishes the means of testing for error conditions after a error-sensitive reference to the I/O Manager.

The function reference is

```
IERR = LMERCD (IERR)
```

If an error condition has been detected in the previous IOM operation, a *nonzero value* is returned as both argument and function value. The double setting facilitates the use of LMERCD in conditional branching statements such as

```
IF (LMERCD(KODE) .NE. 0) CALL ERROR (KODE)
```

REMARK 6.6

LMERCD serves both the IOM and EZ-GAL levels of NICE-DMS. It thus absorbs the function of LMIOER, which was designed to retrieve IOM error codes only.

REMARK 6.7

There is no longer any significant correlation between the error code and a specific error type. On the contrary, the relation will frequently vary as new error conditions are introduced in NICE-DMS, because these are internally sorted (by an *ad-hoc* table-building program) alphabetically on the error key. The error code serves only two purposes: indicates the presence of error by a nonzero value, and works as a "hook" for retrieving error keys and messages through DMEINF (§6.5).

§6.5 EXTRACT ERROR INFORMATION: DMEINF

Entry point DMEINF is used to extract the error key and error message, given the error code.

The calling sequence is:

```
CALL DMEINF (IERR, EMSG, K)
```

where the input is:

IERR	Error code returned by LMERCD.

and the outputs are:

EMSG	A character string that receives the error key in its first 4 locations, followed by a comma and a diagnostic message. The total length of the text string is returned in K. If IERR is zero or is not a proper error code, ESMG is blanked and K set to zero.
K	The length of the message returned in EMSG. If the passed length of EMSG is insufficient to hold the whole message, it is truncated to that value, and K set to LEN(EMSG).

REMARK 6.8

In most cases the user program will be interested only in retrieving and testing the *error key*. The following illustrates a typical construction that tests for a device-open error.

```
CHARACTER*4 KEY
.....
CALL DMDAST (LDI, EDNAME, DDPARS)
IF (LMERCD(IERR) .NE. O) THEN
   CALL DMEINF (IERR, KEY, K)
   IF (KEY .EQ. 'DOPE') THEN
      .....
      .....
   END IF
END IF
```

§6.6 RETRIEVE I/O STATUS CODE: LMIOST

Entry point LMIOST, referenced as an integer function, returns a I/O status code in effect since the last I/O operation.

The function reference is:

```
ICODE = LMIOST (J)
```

where

J	Index to the I/O status array maintained by the I/O manager. Normally J = 1.
LMIOST	J-th entry of the I/O status array.

REMARK 6.9

These values are not only machine-dependent, but depend on whether FORTRAN I/O or Block I/O was used. In the case of FORTRAN I/O, a few things about I/O status codes are described in the FORTRAN-77 standard.

§6.7 DEFUSE FATAL ERRORS: DMEASY

Entry point DMEASY (named after "take it easy") may be used to specify that subsequent fatal errors are to be treated as warning-only.

The calling sequence is:

```
CALL DMEASY (KERR)
```

where

KERR	If KERR > 0, treat next KERR fatal errors as warning only.
	If KERR is zero, the standard error treatment of fatal errors is enforced.
	If KERR < 0, treat next |KERR| fatal errors as warning-only *and* suppress all diagnostic messages. For experienced programmers only.

REMARK 6.10

Each entry to DMSERR counts as one error for the purposes of decrementing KERR.

REMARK 6.11

This entry point is primarily useful for batch runs.

REMARK 6.12

The treatment of catastrophic error conditions is not affected.

REMARK 6.13

DMEASY supersedes one of the functions of an earlier "error handling" entry point DMHAST.

§6.8 SPECIFY ERROR TERMINATOR: DMETER

Entry point DMETER may be called to specify an error termination routine to be called in the event of a fatal error termination.

The calling sequence is:

```
CALL DMETER (UPGERR)
```

where UPGERR is the name of the error termination routine. This name must be declared EXTERNAL in the subprogram that calls UPGERR.

In the event of a fatal error condition, DMSERR calls DMFATE, which checks whether an error-termination routine has been specified using DMETER. If so, it issues the equivalent of the calls

```
CALL UPGERR ('NICE-DMS', EKEY)
```

where EKEY is the error key.

REMARK 6.14

UPGERR must not execute a RETURN. It will be futile, anyway, as the next statement in DMFATE is a call to unconditionally abort the run.

REMARK 6.15

UPGERR should not call DMSERR or DMFATE.

REMARK 6.16

DMETER supersedes functions of the obsolete entry points DMHAST and DMTERM.

§6.9 TAKE FATAL ERROR EXIT: DMFATE

In the event of a fatal error condition, DMSERR eventually calls subroutine DMFATE. This reference is described here, as it may occasionally be useful for inclusion at the user program level.

The calling sequence is:

```
CALL DMFATE (NAME, EKEY)
```

where NAME is the name of the calling package, and EKEY an error key.

If an error-termination routine has been specified through DMETER (§6.8), DMFATE cleverly engineers a transfer to it.

§6.10 PRINT ERROR TRACE STACK: DMPETS

The error trace stack (ETS) of NICE-DMS may be displayed through entry point DM-PETS.

The calling sequence is:

```
CALL DMPETS (-2, PRTFIL)
```

where PRTFIL is the number of the logical unit that is to receive the print output (usually 6). Other values of the first argument are reserved for internal use by NICE-DMS.

7
References

1. Felippa, C. A.: Architecture of a Distributed Analysis Network for Computational Mechanics, *Computers and Structures*, **13**, pp. 405-413, 1981.

2. Wright, M. A., Regelbrugge, M. E., and Felippa, C. A.: *The Computational Structural Mechanics Testbed Architecture Volume IV - The Global-Database Manager GAL-DBM*, NASA CR 178387, January 1989.

3. Felippa, C. A.: *The Computational Structural Mechanics Testbed Architecture Volume I - The Language*, NASA CR 178384, December 1988.

4. Felippa, C. A.: *The Computational Structural Mechanics Testbed Architecture Volume II - Directives*, NASA CR 178385, February 1989.

5. Felippa, C. A.: *The Computational Structural Mechanics Testbed Architecture Volume III - The Interface*, NASA CR 178386, December 1988.

6. Univac 1100 Series Operating System Programmer's Reference Manual, UP-4144, Rev. 3, Sperry-Rand Univac, 1973.

7. SCOPE 3.4 Reference Manual, Cybernet Services, Control Data Corporation, September 1977.

8. NOS Version 1 Reference Manual (2 vols), Control Data Corporation, December 1976.

9. VAX/VMS Command Language Users Guide, Digital Equipment Corporation, Maynard, Massachusetts, 1988.

10. Felippa, C. A.: *Utilities for Master Source Code Distribution: MAX and Friends*, NASA CR 178383, October 1988.

A
COMPILATION INSTRUCTIONS

§A.1 IOM Compilation on VAX under VMS

To compile the I/O Manager on a VAX running under VAX/VMS, you start from three Master Source Code (MSC) files:

1. A FORTRAN Procedure file, assumed to be FORPRC.MSC.

2. A FORTRAN-source file, assumed to be DMGASP.VAX.

3. If Block I/O is wanted, assembly-language source file BIOSUP.MAR.

To extract FORTRAN-compilable versions, you also need the Master Distributor MAX [10]. In what follows it is assumed that MAX is installed on the VAX system, and that you are thoroughly familiar with the its use.

The first step is to *split* FORTRAN-INCLUDE files off from the master procedure file, FORPRC.MSC. Assume that FORPRC.MSC resides in directory [USER.INCLUDE]. Then

```
$ ASSIGN [USER.INCLUDE] PROC:
$ SET DEF PROC:
$ MAX/F/L/SIC <FORPRC.MSC >/INC
```

where use of the logical name PROC is mandatory.

The second step is to extract and compile the FORTRAN version for the VAX. Assume that the Master-Source-Code file, DMGASP.VAX, resides in the directory [USER.IOMANAGER], and the object code is to be inserted in object library [USER.LIBRARY]NICE.OLB:

```
$ SET DEF [USER.IOMANAGER]
$ MAX/F <DMGASP.VAX >.FOR BIO PIO TRACE /L
$ FOR DMGASP
$ LIB [USER.LIBRARY]NICE DMGASP
$ DEL DMGASP.FOR.*, DMGASP.OBJ.*
```

In the MAX command, distribution keys BIO and PIO generate Block I/O and Paged I/O capabilities, respectively, in the DMGASP.FOR file. If the key BIO is omitted, Block I/O capabilities are not included. If the key PIO is omitted, Paged I/O capabilities are not included.

The third and final step is only required if you specified Block I/O capabilities. MACRO source file BIOSUP.MAR has to be assembled and the output inserted in the object library:

```
$ MAC BIOSUP
$ LIB [USER.LIBRARY]NICE BIOSUP
$ DEL BIOSUP.OBJ.*
```

B

GLOSSARY

The following quick-reference list collects terms and acronyms that often appear in the present document.

Access method	The set of procedures for accessing and transferring data structures from a residence medium to another. In the literature, the term is often used in relation to stored databases.
Addressing	The procedure by which a storage address at which a subsequent activity is to take place is specified.
Auxiliary storage	Storage facilities of lower cost and slower access than main storage; generally connected to the central processor by data channels.
Block	A generic term that denotes a string of storage objects such as characters, words, PRUs, etc., which are considered as a storage unit for some purpose.
Block I/O	An Input-Output process that involves direct (unbuffered) transfers of blocks of data between main storage and a disk volume. Available from many operating systems through special service entry points.
Catalogued file	Univac terminology for permanent files whose names are maintained by the system on a Master File Directory.
Closing (a device)	See *device closing.*
Core device	A word-addressable, scratch device that resides on blank-common storage.
Current device location	A storage address maintained by the I/O Manager for each active logical device, and which identifies the location at which the next read or write operation is to take place.
Data	Information recorded on a storage device.
Database	An organized collection of operational data needed for the completion of an activity. The term is usually reserved for activities at the task or project level.
Database manager (DBM)	A data management system that interfaces a database with its user environment.
Data library	A named partition of a database.

Data management system	A software module that centralizes activities pertaining to the manipulation of a class of data structures.
Data manager	The decision-making component of a data management system.
Dataset	A record, or set of records, that is a named element of a data library.
Data space	See *storage space.*
Data structure	A set of interrelated data objects viewed as a single logical entity.
Device	See *input/output device, logical device.*
Device closing	A process by which facilities assigned to a logical device are released (returned) to the operating system. A freed device is *inactive.* If the device was opened as scratch, its contents disappear.
Device declaration	See *device open*
Device opening	A process by which facilities assigned to a logical device are requested from the operating system. An open device is said to be *active.*
Device option index (OPTX)	An index that characterizes permanency and accessibility attributes of a logical device when the device is opened (open time).
Device type index (TYPEX)	An index that describes residence and granularity attributes of a logical device at open time.
Direct-access storage	A type of storage that is capable of processing data at separate locations without passing over the intervening data. Also known as *random-access storage*, connoting the property that items of data can be stored or retrieved efficiently in a random order.
DMGASP	The particular I/O Manager described in this document.
Dynamic	A qualifier applied to certain actions, such as the declaration and freeing of storage facilities, which are performed on command from a running program. (Contrast to *static*, in which such actions are performed before or after running the program.)

Extent	A contiguously addressed storage region; also the size of any such region.
External device name	The symbolic identifier of a logical device given to the I/O manager by the user program. For disk-resident devices, this identifier contains the external file name, and often *is* simply the file name. The external device name is only used at device declaration time; from then on the device is identified by its Logical Device Index.
External file name	The identifier by which the residence of a file is specified to the operating system.
External PRU	The physical record unit by which the user of the I/O manager addresses a direct-access logical device.
Facilities	Storage equipment available at a computer installation.
File	See *logical file, physical file.*
File name	The identifier(s) by which a logical file is known to the operating system.
Global database	A database residing on permanent storage, and which is accessible by a network of communicating programs.
Hardware PRU	A physical record unit that corresponds directly to the mechanical and/or electronic characteristics of the storage medium.
Information	Quantifiable knowledge.
Information structure	An organized collection of information viewed as a logical entity.
I/O Device	A storage device connected to the central processor by a data channel.
I/O Manager (IOM)	The component of a multilevel data management system that is responsible for the access method.
Internal file name	The identifier by which a file structure is referenced by a running program. It is linked to the external file name (and the Logical Device Index) at open time.
Internal PRU	The PRU size used by the I/O manager for requesting physical-record transfers. For Block I/O devices, it co-

	incides with the hardware PRU. For FORTRAN I/O devices, it is the Fixed Record Length declared for direct-access devices.
Local database	A database attached to a running program, and which disappears when the program stops.
Local file	CDC terminology for *temporary file.*
Location	An addressable component of a storage device.
Logical file	The description mechanism by which logical devices residing on auxiliary storage are managed by the operating system.
Logical device	A partition of an I/O device that is managed as a logical entity for resource-allocation and administration purposes. For auxiliary storage devices, the term is equivalent to *logical file.*
Logical Device Index (LDI)	An integer that identifies a logical device entered in the Logical Device Table (LDT) of the I/O manager.
Logical Device Table (LDT)	A table of logical devices maintained by the I/O Manager.
Logical name	DEC term for *internal file name.*
Logical record	A record structure as seen by the applications programmer.
Main storage	Random-access storage facilities hardwired to the central processing unit, and referenceable by machine-code addresses.
Manager	A software element that is primarily engaged in the administration of computing resources.
Mass storage	CDC term for online, large-capacity auxiliary storage facilities allocatable for public use.
Online storage	Storage under direct control of the central processing unit.
Open (a device)	See *device opening.*
Page Buffer Pool (PBP)	An area of main storage set aside for the realization of Paged I/O.

Paged I/O	An implementation of buffered I/O in which data transfers between the user-program workspace and an auxiliary storage device take into account the presence of a Page Buffer Pool in main storage.
Permanent file	A file structure that survives the execution of the process that created or modified it. A permanent file exists until it is specifically deleted by its owner, or (if lapsed) by the operating system.
Permanent file name (PFN)	CDC term for *external file name* of a catalogued file.
Physical device name	DEC term for *external file name.*
Physical record	A record structure as presented to the operating system services.
Physical record unit (PRU)	The addressing unit "granule" for direct-access devices. Varies according to usage level: see *external PRU, hardware PRU, internal PRU, sector.*
Positioning (a device)	The insertion of a storage address into the Logical Device Table to update the current device location.
Random-access storage	See *direct-access storage.*
Record	A set of data objects characterized by physical adjacency, which constitutes the basic transaction unit in the transmission of data between main and auxiliary storage.
Scratch file	A file structure that disappears when the process that created it stops, or when the file is explicitly closed.
Sector	The smallest addressable unit by the operating system on a rotating direct-access storage device such as a drum or disk. It may be a true equipment characteristic (in which case it coincides with a hardware PRU) or the result of simulation by the operating system.
Sequential-access device	A type of storage in which the data can be accessed only by following the order in which it was stored.
Storage	Any device that is capable of retaining information over a period of time and of delivering it on request.
Storage address	A label, name or number that identifies the place at which data are recorded on a storage device.

Storage device	A subset of the storage facilities that is treated as an operational entity for purposes of allocating or releasing storage resources during the execution of a run or process.
Storage peripheral	A readily detachable part of the storage facilities; for example a magnetic tape unit or a removable disk volume.
Temporary file	A file structure that disappears when the job that created it terminates, or when its facilities are released. (On many systems, temporary and scratch files are undistinguishable.)
Track	The portion of a mechanical storage device such a drum, disk, or tape, which is accessible to a given read/write station.
Unit	See *logical unit, storage unit.*
Volume	The storage space associated with a separable segment of the storage facilities on a one-to-one basis; *e.g.*, a magnetic tape reel, mountable cartridge or disk drive.
Word	The standard main-storage allocation unit for numeric data. Conventionally, a word holds a single-precision floating-point value.

THIS PAGE LEFT BLANK INTENTIONALLY.

C
PAGED I/O PERFORMANCE

§C.1 TO PAGE OR NOT TO PAGE

The present I/O Manager offers the programmer a choice between Paged I/O and Unpaged I/O for *word addressable* devices resident on auxiliary storage.

The choice has no effect on device contents: two files properly generated by identical runs, one with Paged I/O and one with Unpaged I/O, will be identical to the last bit. It is also transparent as far as device accessing.

The choice does affect I/O *performance*. The purpose of this Appendix is to examine issues pro and con, and to provide comparative performance data gathered on the VAX 11-780.

§C.2 TRANSPARENCY

How "transparent" is Paged I/O from a programming standpoint? The issue has two sides: effects on the user program code, and effects on device contents.

User Program Code. To accommodate Paged I/O, the user-program developer has to make sure of three things:

1. Declare a Page Buffer Pool (PBP) area through DMPOOL (§4.6).

2. Provide a scheme for optionally setting the fourth item of the DDPARS array to a negative value when calling DMOPEN or DMDAST (§3.1).

3. Provide robust mechanisms to ensure integrity of Paged devices that have been written to and reside on permanent files (§2.4.3-2.4.4). Two examples: periodic calls to DMFLUB (§4.3); closing all devices in an error-termination routine declared through DMETER (§6.8).

Fortunately, these are minor additions to the program source code. The majority of the program source code (global and local data management, record transfers, ...) is not affected.

Device Contents. As far as what's in the device, there are no changes, since Paged I/O pays no attention to device contents. You are therefore free to do things like the following. Open a new device with Unpaged I/O, write to it, close it. Reopen read-only with Paged I/O, read records. Close again, reopen Unpaged, write to it, and so on. Such open-close sequences can take place during the execution of a single processor or macroprocessor, or multiple executions of several processors.

The only restrictions are: the device must be opened under either option, be word-addressable, and if a permanent-file Paged device is modified, the PBP must be appropriately flushed before the run terminates.

§C.3 DISADVANTAGES OF PAGED I/O

The following general disadvantages of Paged I/O are annotated here for later reference.

1. Paged I/O is inherently riskier than Unpaged I/O. Having an intermediate record buffer is detrimental to data integrity in two ways: the PBP may be accidentally overwritten, and modified pages may be lost if the run aborts. (The latter consideration is irrelevant, however, for read-only or scratch devices.)

2. Paged I/O performance is sensitive to more environmental parameters. There are more decisions to be made (*e.g.*, page size, PBP length) and these may be machine- and/or problem-dependent.

3. Declaring the Page Buffer Pool ties up main storage. This is more important on non-virtual machines with strict limitations on physical main storage, but unimportant on virtual-memory machines.

§C.4 PERFORMANCE PARAMETERS

Two key parameters that affect Paged I/O performance are:

s	mean size of records transferred.
d	mean "device location distance" traveled between successive record transfers.

The mean record size, s, has significant impact on performance. The smaller the value of s, the better the performance of Paged I/O, as this increases the likelihood of "page hits", (*i.e.*, finding data in the PBP when reading or collecting small records into bigger ones when writing). But how small is small? This has to be answered by performance tests on actual computers.

The mean-distance-between-records, d, characterizes the "locality" of device references. Smaller values of d favor Paged I/O because the PBP fragmentation is reduced, with a consequent reduction in page-faulting probability.

Three other factors that affect performance are: page size p, number of pages n in the PBP, and number of Paged I/O devices simultaneously active. If the page size and PBP length is chosen as recommended in §4.6, these three factors have marginal effect.

§C.5 A PERFORMANCE TEST

A fairly comprehensive performance test contrasting Paged and Unpaged I/O was performed on a VAX 11-780 running under VAX/VMS 2.5. The test proceeded as follows.

1. Parameters p (page size) and buffer length $b = pn$ (where n is the number of pages in the PBP) were selected and specified using DMPOOL.

2. A fairly large number of words W (typically 50000 or 100000) was written to a Paged I/O scratch file in many combinations of record sizes s and distances d. (A statistically correct d-sequence was produced by a random number generator.) The same number of words was then read back from the file, which was closed after each cycle. The CPU time used and the number of system I/O accesses were recorded for each write-read cycle.

4. The test was repeated with Unpaged I/O and the same set of sample record sizes, but with d set equal to s for convenience, as Unpaged I/O is insensitive to d. CPU times and system I/O accesses were recorded.

The performance ratio based on spent CPU time is

$$\rho(s,d) = \frac{t\ U(s)}{t\ P(s,d)}$$

where $t\ P$ and $t\ U$ denote CPU times for the Paged and Unpaged I/O write-read cycles, respectively. $\rho > 1$ favors Paged I/O, while $\rho < 1$ favors Unpaged I/O. A similar ratio was computed from the I/O-access count data, but the values were more erratic. Consequently, the CPU time ratio ρ was selected as the basis for the comparison presented here.

Figures C.1 through C.3 display log-log plots in the (s, ρ) axes for the following three cases:

Figure	p (words)	n (words)	b (words)	Access Method
C.1	128	32	4096	Block I/O
C.2	512	8	4096	Block I/O
C.3	128	32	4096	FORTRAN I/O

The four curves labeled **A** through **D** pertain to different selections of the mean record distance, d:

Curve	Distance	Comments
A	d = s/5	Highly clustered records
B	d = s	Sequential-access like records
C	d = 5s	Moderately spread records
D	d = 25s	Highly spread records

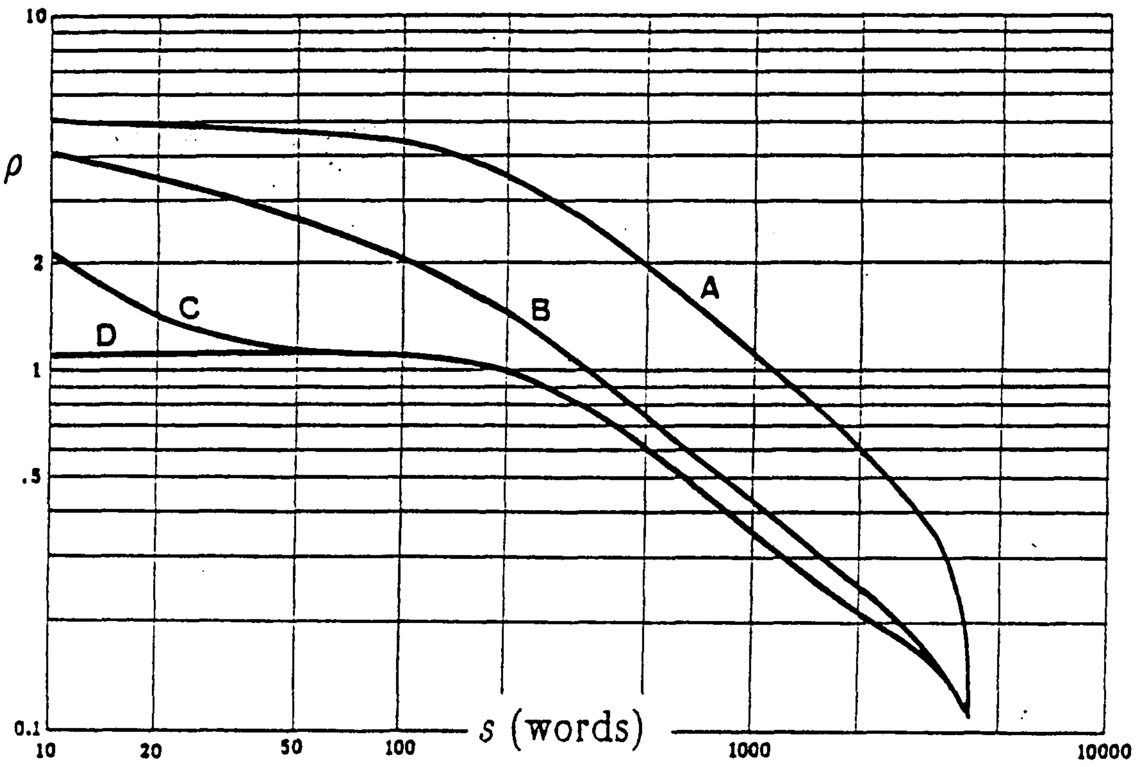


Figure C.1. CPU-time Ratio $\rho(s,d)$ for
Block I/O Device, $p = 128$, $b = 4096$

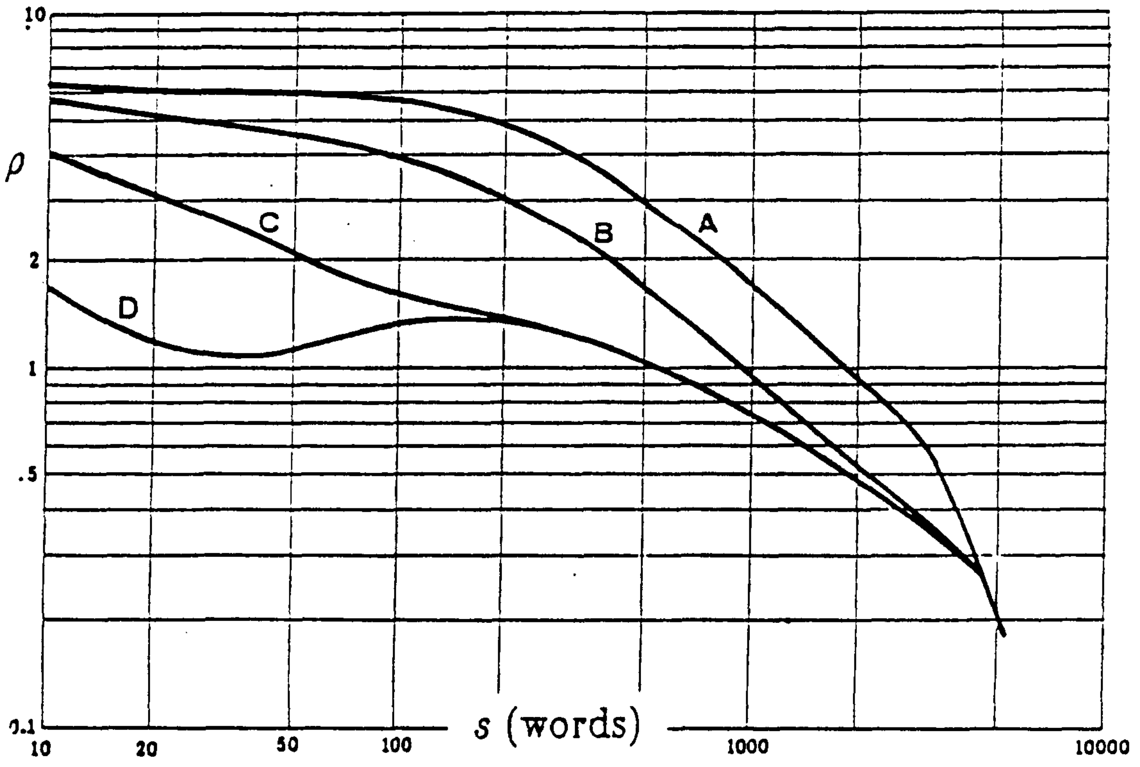


Figure C.2. CPU-time Ratio $\rho(s,d)$ for
Block I/O Device, $p = 512$, $b = 4096$

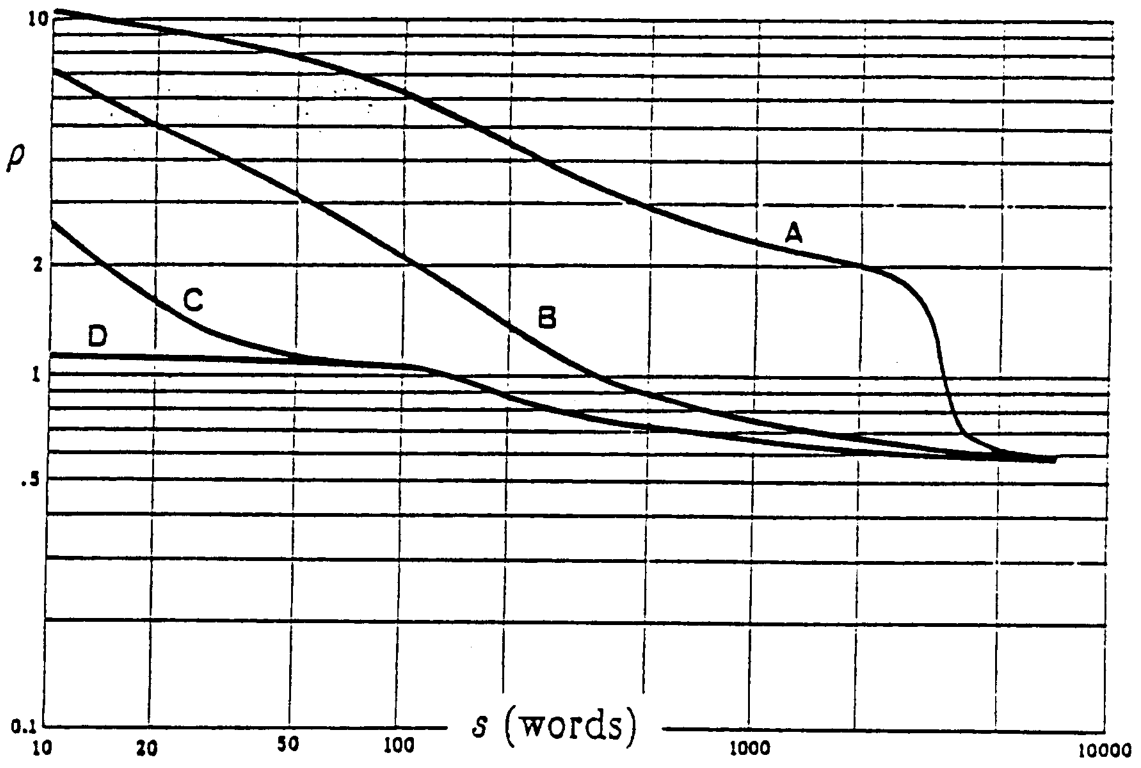


Figure C.3. CPU-time Ratio $\rho(s,d)$ for FORTRAN I/O Device, $p = 128$, $b = 4096$

Figure C.1 (Block I/O, page size $p = 128$) confirms what has been said about the effect of s and d. Note also the sudden degradation of Paged I/O as s gets near the buffer size $b = 4096$; each record transaction then forcibly empties the PBP, which causes thrashing.

Figure C.2 shows the effect of increasing the page size p to 512 words, while keeping $b = 4096$. This has a beneficial effect on Paged I/O performance over most of the s range. Increasing p further (to 2048, while boosting b to 16384) had only marginal effect for $s < 2000$.

Figure C.3 shows the effect of using direct-access FORTRAN I/O with an internal PRU of 128 words (TYPEX = 4, cf. Table 2.2). In this case $p = 128$ had the best performance, no doubt because of the double buffering (cf. §2.4.4). Paged I/O performance gains for small clustered records ($s < 100, d \leq s$) appear fairly impressive. However, it must be kept in mind that FORTRAN I/O is roughly 10-15 times slower than Block I/O on the VAX.

§C.6 RECOMMENDATIONS

For certain NICE Processors the choice between Paged and Unpaged I/O is clear-cut. Two examples.

Interactive Pre- and Post-Processors. These naturally entail processing of many small, space-clustered records (typically $s < 100$, $d \leq 2s$). Paged I/O is the obvious choice.

Large-Scale Matrix Processors. These typically involve large records, say $s = 1000$ to 100000 words, so choosing Paged I/O would be impractical.

But there are many processors for which the choice is unclear; for example element-matrix processors in finite element analysis. Here record sizes and clustering may be heavily problem dependent. Also, performance crossover points are computer dependent.

For the doubtful cases, an *in vivo* performance test is the only answer. Here is where the transparency of Paged I/O on program and data structures pay off. For example, Paged I/O might be profitable on a fast-CPU/slow-I/O machine (*e.g.*, CDC, Cray) while being unprofitable on a VAX or IBM. Being able to turn it on or off selectively (even from device to device) without touching the Master Source Code is a definite plus.

Should the decision be marginal, *Unpaged I/O should be preferred.* Why? Unpaged I/O is safer, ties up less memory, and is less sensitive to environmental parameters. This recommendation includes all processors that are CPU bound.

THIS PAGE LEFT BLANK INTENTIONALLY.

D

MISCELLANEOUS UTILITIES

The I/O Manager package includes some miscellaneous utilities which are not directly related to Input-Output activities. These are described below as they have proven generally useful.

§D.1 CONVERT CHARACTER TO HOLLERITH: CC2H

CC2H converts a FORTRAN 77 character string to a Hollerith string.

The calling sequence is:

```
CALL CC2H (C, H, N)
```

where

C	Source character string.
H	Receiving Hollerith string (typed integer, floating-point, or logical in the calling program).
N	Number of characters to be moved. No operation if N $\leq$ 0.

REMARK D.1

Characters are stored in H beginning at its leftmost location. If H is of INTEGER or REAL type, this is necessarily word-aligned. CC2H does *not* right blankfill H, however.

REMARK D.2

This utility is handy for writing character records through DMWRIT or DMWAST (§3.4).

REMARK D.3

The implementation of CC2H has turned out to be surprisingly machine-dependent: so far, five versions had to be written for five computers. The VAX implementation, which takes advantage of the LOGICAL*1 data type, is the simplest and most efficient:

```
SUBROUTINE CC2H (C, H, N)
CHARACTER C(*)
LOGICAL*1 H(*)
DO 1000 J = 1,N
   H(J) = ICHAR(C(J))
1000 CONTINUE
RETURN
END
```

But this doesn't work on other FORTRAN 77 compilers. On word-addressable machines (Univac, CDC), the LOGICAL*1 type does not exist, so one is forced to do masking and Boolean operations on H.

Byte-addressable machines generally supply the LOGICAL*1 type, but the compiler may forbid type conversions. For example, IBM's VS-FORTRAN offers LOGICAL*1, but forbids storing an integer value into it. The internal-file transfer feature of FORTRAN 77 provides a workable although inefficient solution:

```
SUBROUTINE CC2H (C, H, N)
CHARACTER C(*)
LOGICAL*1 H(*)
M1 = 0
DO 1000 K = 1,(N+127)/128
   M2 = MIN(M1+N-128*(K-1),M1+128)
   READ (C(M1+1:M2),'(128A1)'), (H(I),I=M1+1,M2)
   M1 = M2
1000 CONTINUE
RETURN
END
```

§D.2 CONVERT HOLLERITH TO CHARACTER: CH2C

CH2C converts a Hollerith string to a FORTRAN-77 character string.

The calling sequence is:

```
CALL CH2C (H, C, N)
```

where

H	Receiving Hollerith string (typed integer, floating-point, or logical in the calling program).
C	Destination character string.
N	Number of characters to be moved. No operation if $N \leq 0$.

REMARK D.4

This utility is handy for reading character records through DMREAD or DMRAST (§3.5).

REMARK D.5

Remark D.3 about machine dependencies also applies to CH2C. Again the VAX implementation is the simplest: the fifth line changes to C(J) = CHAR(H(J)).

§D.3 FIND BATCH OR INTERACTIVE: FBI

Entry point FBI finds whether the run is batch or interactive.

The calling sequence is:

```
CALL FBI (RUNMOD)
```

where

RUNMOD	An integer variable that receives a run mode indicator.
	0: batch mode.
	1: interactive mode.

REMARK D.6

This utility is highly machine dependent, because FBI must query the operating system. Some operating systems do not provide information of this nature; in such a case RUNMOD = 0 is returned.

§D.4 GET BLANK-COMMON LOCATION OF ARGUMENT: LOCBCW

Function LOCBCW computes and returns the blank-common *word* address of its argument.

The function reference is:

```
ILOC = LOCBCW (ARG)
```

where

LOCBCW	Blank-common *word* address of the function argument ARG.

REMARK D.7

ARG does not have to be physically in blank common (but see next Remark).

REMARK D.8

On byte-addressable machines, the result may be off by one unless ARG happen to be exactly word-aligned with the start of blank common.

REMARK D.9

The implementation of LOCBCW is machine-dependent, as accessing the internal address of an argument is not an operation defined in standard FORTRAN.

THIS PAGE LEFT BLANK INTENTIONALLY.

E

INDEX

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No. NASA CR-178388	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle The Computational Structural Mechanics Testbed Architecture Volume V - The Input-Output Manager DMGASP		5. Report Date March 1989
		6. Performing Organization Code
7. Author(s) Carlos A. Felippa		8. Performing Organization Report No. LMSC-D878511
9. Performing Organization Name and Address Lockheed Missiles and Space Company, Inc. Research and Development Division 3251 Hanover Street Palo Alto, California 94304		10. Work Unit No. 505-63-01-10
		11. Contract or Grant No. NAS1-18444
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225		13. Type of Report and Period Covered Contractor Report
		14. Sponsoring Agency Code

15. Supplementary Notes
Carlos A. Felippa, Center for Space Structures and Controls,
University of Colorado, Boulder, CO 80309-0429

Langley Technical Monitor: W. Jefferson Stroud

16. Abstract
This is the fifth of a set of five volumes which describe the software architecture for the Computational Structural Mechanics Testbed. Derived from NICE, an integrated software system developed at Lockheed Palo Alto Research Laboratory, the architecture is composed of the command language (CLAMP), the command language interpreter (CLIP), and the data manager (GAL). Volumes I, II, and III (NASA CR's 178384, 178385, and 178386, respectively) describe CLAMP and CLIP and the CLIP-processor interface. Volumes IV and V (NASA CR's 178387 and 178388, respectively) describe GAL and its low-level I/O. CLAMP, an acronym for Command Language for Applied Mechanics Processors, is designed to control the flow of execution of processors written for NICE. Volume V describes the low-level data management component of the NICE software. It is intended only for advanced programmers involved in maintenance of the software.

17. Key Words (Suggested by Authors(s)) Structural analysis software Command language interface software Data management software		18. Distribution Statement Unclassified—Unlimited Subject Category 39	
19. Security Classif.(of this report) Unclassified	20. Security Classif.(of this page) Unclassified	21. No. of Pages 157	22. Price A08

NASA FORM 1626 OCT 86
For sale by the National Technical Information Service, Springfield, Virginia 22161-2171