IAC Doc. No. SG-I1000-0000-C

Rev. 7-1-73

49

SYSTEM GUIDE for the ILLIAC IV USER

Institute for Advanced Computation

Ames Research Center

Moffett Field, California 94035

SYSTEM GUIDE FOR THE ILLIAC IV USER

CONTENTS

IAC Doc. No. SG-I1000-0000-C
Rev. 7-1-73

SECTION 1

INTRODUCTION

SECTION 1

INTRODUCTION

This guide, the first in a series written for the ILLIAC IV System user, describes the necessary procedures for using the ILLIAC IV: how to gain access to the System, how to correctly construct a job to run on the System and to submit the job for processing, and how to transfer data between the user's entry location and the ILLIAC IV System.

A potential user of the ILLIAC IV will generally have three steps to go through to successfully use the ILLIAC IV System. He must understand the ILLIAC IV System resources and characteristics sufficiently to determine their suitability to his problem, design and program his problem solution, and interact with the ILLIAC IV System to guide execution of his ILLIAC IV programs. This guide is directed toward the third step in this sequence, user interaction and control of the ILLIAC IV System. It includes information, conventions, and suggested procedures to process work successfully on the ILLIAC IV System.

1.1 CONTROL LANGUAGE FOR THE ILLIAC IV SYSTEM

The primary tool provided to the user to interact with the ILLIAC IV System is ACL (A Control Language). Access is gained to the ILLIAC IV System through the ARPA Network, and interaction with the ILLIAC IV System begins through ACL.

Each ACL statement initiates a subsystem that performs a selected function for the user. All of the requested processing is directed by the user through ACL statements. Accordingly, this System User's Guide is oriented around ACL and the subsystems it initiates.

1.2 <u>SUGGESTED USE OF THIS GUIDE</u>

The ILLIAC IV System is a complex set of computing resources provided to its community of users. ACL, the bridge between the user and the System, will evolve as system resources are implemented, providing and supporting new capabilities as they become available.

ACL is presented at two levels in this guide. A basic structure and a basic set of ACL are sufficient for the average user. This basic set of ACL is presented in tutorial fashion in Sections 2 through 5 of this guide. These sections <u>must</u> be understood by all users of the ILLIAC IV System.

The complete set of ACL statements is given in Section 7 and its intended use is for reference purposes. Section 6 provides a broader coverage of the rules of construction of the language, including all of the notational conventions used in Section 7. Section 8 provides examples of some of the more complex capabilities of ACL. It is suggested that the user acquaint himself with this material in conjunction with acquiring a thorough understanding of Sections 2 through 5.

This guide begins in Section 2 with a brief overview of the ILLIAC IV System, intended to give the user some feeling for the resources available and the structure of the ILLIAC IV System. Further documentation on the ILLIAC IV System is listed in the Bibliography, Appendix C.

Appendix B is a fairly extensive glossary of the special terminology used in this guide. Most of the glossary entries contain references to specific sections of the guide where detailed explanations of the terms can be found.

SECTION 2

SYSTEM OVERVIEW

SECTION 2

SYSTEM OVERVIEW

CONTENTS

SECTION 2

SYSTEM OVERVIEW

The ILLIAC IV System contains a set of resources available to the system users. This overview is oriented to a user whose objective is to run programs on one of these resources, the ILLIAC IV. Users whose central interest is directed toward resources other than the ILLIAC IV may find this description of the System not suited to their purposes.

2.1 LOCATION AND ACCESS

The ILLIAC IV System is a complex of computing equipment located at NASA-Ames Research Center, Moffett Field, California. The data processing and storage resources of the System are available to remote users through the ARPA Network. ILLIAC IV System users can access the System through terminals connected directly to the ARPA Network or through a remote computer system tied into the ARPA Network (referred to as a Host system). Data transfers to and from the ILLIAC IV System comply with the ARPA Network File Transfer Protocol (FTP).

An ILLIAC IV System user communicates with the System using A Control Language, ACL. ACL consists of statements, formatted according to standard rules, containing both fixed and variable information. The fixed information in an ACL statement identifies the system function the user wishes to initiate, and the variable information provides control directions and data which are passed to the function.

ACL statements may be directly communicated to the System and processed on-line, statement by statement (interactively) or in batch from a user-created file of ACL. A set of ACL statements defines a processing sequence to the System.

2.2 FUNCTIONS PROVIDED BY THE ILLIAC IV SYSTEM

An ILLIAC IV System user sees the System functionally through ACL. In order to properly use ACL, the user must have some knowledge of the System configuration and the available resources. In general terms, the System provides the basic functions of processing, file management, and file storage. More specifically, the System provides the functions described in this guide which are initiated and directed by the user through ACL. Examples of these functions provided through ACL include language processing, file transfers, and data conversions. The System providing these functions comprises a number of processors, memories, mass storage devices, interfaces, peripherals, and supporting software.

In addition to those functions available to the user through ACL, there are functions provided only to the ILLIAC IV programmer, for example, data transfers between the ILLIAC IV disk and array memories. In order to program the ILLIAC IV, the user must have detailed knowledge of these functions. This information is included in the ILLIAC IV Programmer's Guide and supporting documentation; the overview presented here will not attempt to provide this level of description.

A program designed and written for the ILLIAC IV may be translated, stored, loaded, and run through ACL; data files can be copied, converted, and transferred to the ILLIAC IV for use in conjunction with the user's program, again through ACL. In these and other ACL sequences, certain system functions are involved that are not directly available to the user either through ACL or the programming languages — for example, resource scheduling and management. The ILLIAC IV System has been designed and implemented so that details of these system functions need not be of concern to the user.

This overview of the System is presented in functional terms since this is the fundamental concern of the user. Where it is necessary to understand a physical device, its characteristics, or usage, the information is given here and in subsequent sections of this guide.

2.3 ILLIAC IV SYSTEM RESOURCES

There are four principal resources in the ILLIAC IV System: (1) the central system, (2) the ILLIAC IV, (3) the file storage subsystem, and (4) the B6700 computer. See Figure 2-1. Briefly, the functions performed by these resources are as follows:

- The central system provides the user-ACL interface, performs overall system scheduling and resource management, provides and manages the file system, and performs operating system functions for the ILLIAC IV.
- The ILLIAC IV is the major data processing resource in the System and is dedicated to the execution of user programs.
- The file storage subsystem consists of a hierarchy of storage devices ranging from central memory to the UNICON mass storage laser memory device.
- The B6700 provides the user with utility services including the ASK assembler, the GLYPNIR compiler, and the ILLIAC IV Simulator, SSK.

These are the resources in the System which provide services directly to the user. Each of them is a subsystem of devices and supporting software. In general, the user should understand the functions provided by each subsystem, but is not required to understand the details of devices to use ACL.

2.3.1 THE CENTRAL SYSTEM

The central system consists of processors, memory, and interface devices to the other system resources and to the communications subsystem (see Figure 2-1). Functionally, the central system includes the ACL executive and executes all of the ACL subsystems, except for the program preparation utilities provided by the B6700. In order to maximize its processing efficiency, the ILLIAC IV itself has no operating system. The central system provides operating system functions for the ILLIAC IV. These functions are distributed among several processors.

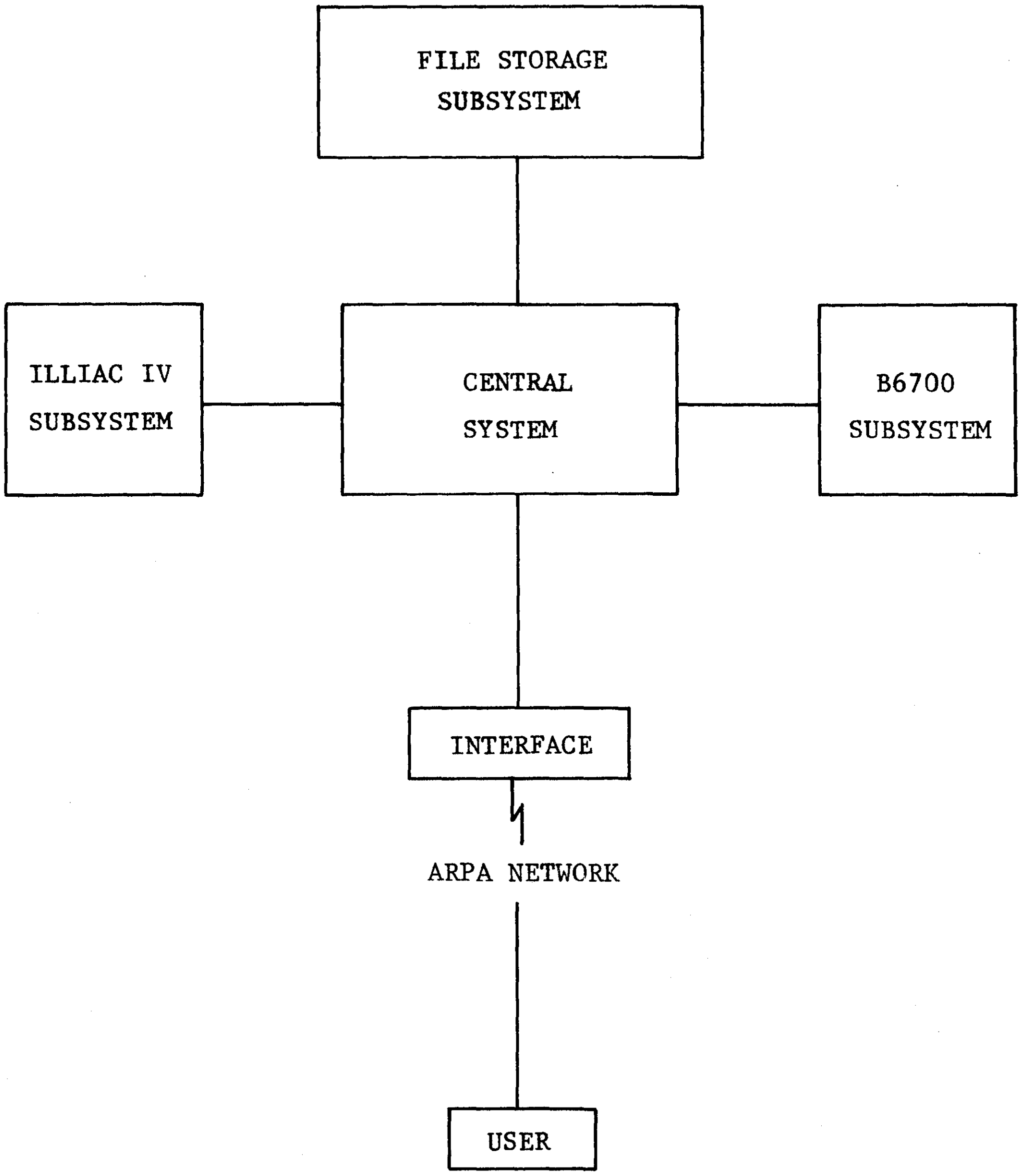


Figure 2-1. Overview of the ILLIAC IV System

The main processor in the central system is a PDP-10. The PDP-10 executes the ACL executive and performs the central management functions of the System, including resource and job scheduling, and logical file management.

Other processors in the central system perform specialized tasks which are transparent to the user. For example, one processor called the MMP manages data transfers between central memory and the ILLIAC IV memory system. It also fields and services ILLIAC IV processor interrupts. The functions of the MMP are described in greater detail in the ILLIAC IV Programmer's Guide. The UNICON memory processor, or UMP, as another example, performs services analogous to the MMP for the UNICON laser memory system.

Other processors in the central system perform communications, diagnostic, and device control functions which will not be described further here.

The central memory provides program and data storage for the central system processors. One of these memory modules, termed the Buffer Input/Output Memory (BIOM), is a 16K (32-bit) word memory which functions as a staging buffer between the central system and the ILLIAC IV memory system. All of the central system processors access central memory and communicate with each other through it. Some of the processors, such as the MMP and the UMP, have small local memories principally for program storage.

2.3.2 THE ILLIAC IV

The major processing resource in the System is the ILLIAC IV (see Figure 2-2), which is dedicated to user program execution. The ILLIAC IV processor is capable of operating in parallel on 64 independent data streams. To achieve this parallelism, the processor structure consists of a control unit (CU) and an array of 64 individual processing elements (PE's). Each ILLIAC IV PE is roughly equivalent to a conventional system's arithmetic unit. The CU decodes each instruction and generates control signals for all of the PE's in the array. Each PE in the array executes the same instruction in parallel, each processing its own data stream.

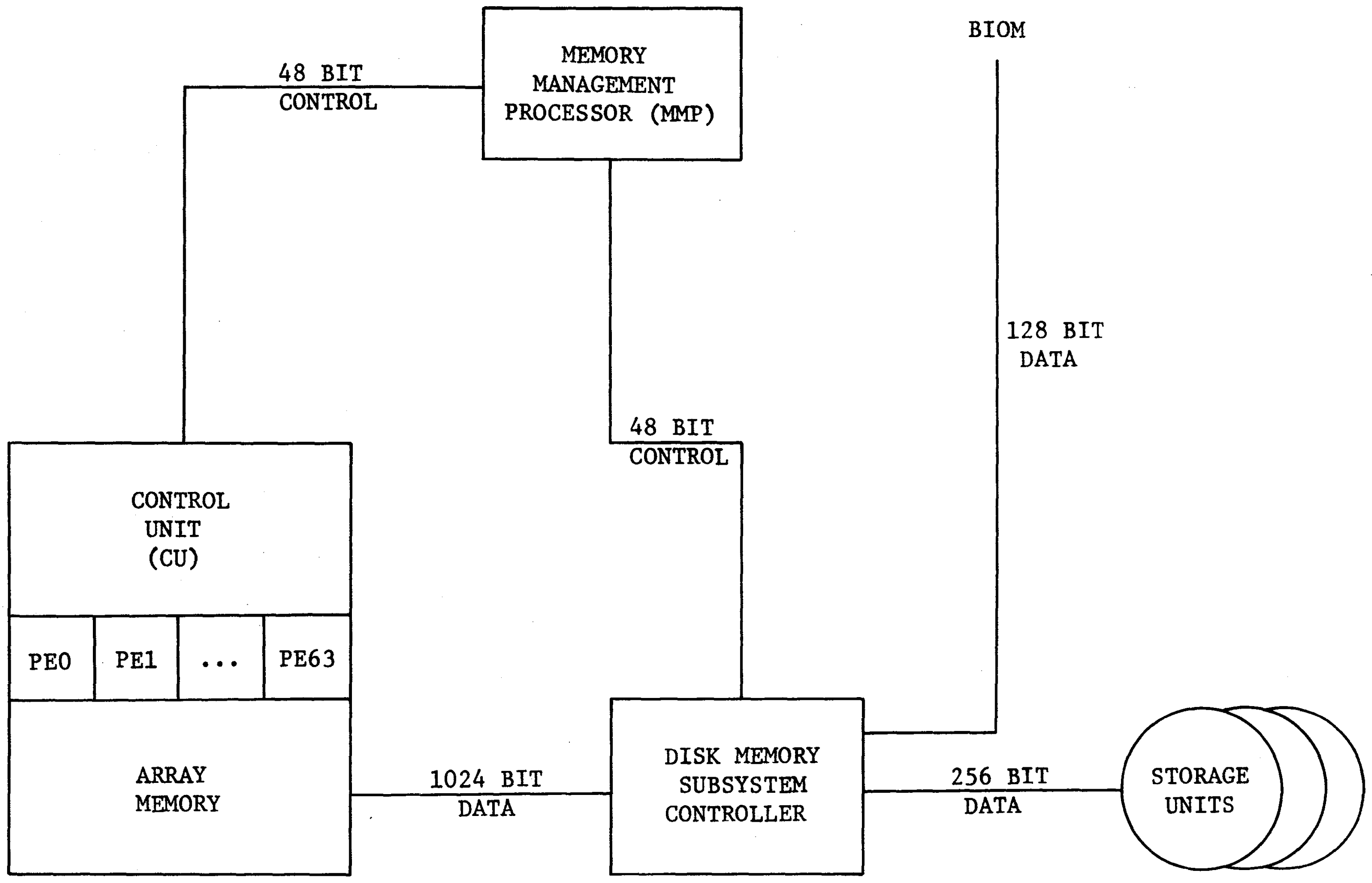


Figure 2-2. ILLIAC IV Block Diagram

Instructions which do not operate on the data streams (such as JUMP) are executed only by the CU and are called CU instructions. Instructions operating on the data streams (for example, arithmetic and logical instructions) are decoded by the CU and control signals are broadcast to the PE's. The CU itself is modularized so that the execution of CU instructions may be overlapped with PE operations.

The instruction execution rate of the ILLIAC IV approaches 3×10^6 operations per second. A high instruction rate is achieved in the PE array not only through parallelism, but also through very fast logic circuits and memory access times. For example, the individual PE's have approximately a 438-nanosecond ADD time and 500-nanosecond MULTIPLY time for full 64-bit operands. The memory cycle time is 313 nanoseconds.

The memory system for the ILLIAC IV includes two main storage devices, the Array Memory and the ILLIAC IV Disk Memory system. Working storage for both instructions and data for the ILLIAC IV Processor is the Array Memory. The Array Memory is a semiconductor memory consisting of 128K (64-bit) or 256K (32-bit) words, with a cycle time of 313 nanoseconds. The Array Memory may be thought of as an array of 64 columns (each associated with a single PE) by 2048 (64-bit) rows. The CU can access the entire Array Memory, while each PE accesses only its associated column.

The ILLIAC IV program is loaded into Array Memory by the ACL RUN subsystem. Data is structured and loaded into the ILLIAC IV memory system by the programmer/user.

The main memory store for the ILLIAC IV is the ILLIAC IV Disk Memory (I4DM) system. The I4DM is a fixed-head rotating disk system with a capacity of about 16×10^6 (64-bit) or 32×10^6 (32-bit) words. The system is composed of 13 disks attached to two controllers (six disks on one controller, seven on the other). The disks rotate synchronously with a 40-millisecond rotation period, and the maximum data transfer rate is about 10^9 bits/second.

Data transfers during program execution between the I4DM and Array Memory are program-initiated. Requests are passed from the ILLIAC IV CU to the central system, where they are executed. Data transfers between the central file system and the I4DM (typically pre- and post-execution) are also under the control of the central system, using the BIOM central memory module as a staging device.

The ACL MOVE subsystem performs the transfers between the central file system and the I4DM. Certain ILLIAC IV memory management functions are also provided in ACL. A more complete discussion of data transfers and the ILLIAC IV memory system is included in Section 5 of this guide.

2.3.3 FILE STORAGE SUBSYSTEM

The file storage subsystem encompasses a series of devices ordered in a hierarchy from central memory to the mass storage laser memory (see Figure 2-1). The devices in this system, in the present implementation, include the following:

- Central Memory
- Swapping Drum
- Buffer Disk
- Laser Memory

All files in the system (both program and data) are composed of pages. Pages are the smallest units of data seen by the file management supervisor of the central system. File pages are moved through the storage hierarchy depending on their activity, space availability, and the total file size. File pages actively being processed by the central system reside in central memory or on the swapping drums. File pages being transferred between the file storage subsystem and the B6700 or the ILLIAC IV transit through central memory and may reside temporarily on the buffer disk. The permanent mass storage repository for user data files is the laser memory.

The UNICON laser memory device has an on-line storage capacity of roughly 700 billion bits which are written permanently onto coated Mylar strips by a laser beam. The access time to a given piece of data on a strip varies depending on the state of the device and can be as long as ten seconds. The transfer rate of the device once it is positioned is roughly four million bits/second. As the Mylar strips become obsolete or are to be archived, they are dismounted and stored. Fresh strips are mounted in their place to provide new areas for writing.

ACL statements are provided to perform user-controlled functions on files; these include CPYNET, MOVE, and COPY (file transfer functions), DEL (delete), and DIR (list the file directory). Files of text may be created interactively using the DED text entry and editing subsystem. The user need not be concerned with how the system manages his files or where they are located in the hierarchy of storage devices, except when he wishes to retrieve an "archived" file from the UNICON, as explained elsewhere in this guide.

2.3.4 THE B6700 COMPUTER SYSTEM

The Burroughs B6700 computer performs utility functions augmenting the functions provided by the central system. When an ACL subsystem requires the B6700, it is noted under the statement description in the sections of this guide that follow. The ACL user or ILLIAC IV programmer does not require any additional detail on the B6700.

SECTION 3

INTRODUCTION TO ACL

SECTION 3

INTRODUCTION TO ACL

CONTENTS

SECTION 3

INTRODUCTION TO ACL

ACL (A Control Language) allows users to communicate with the ILLIAC IV System in order to control various user features of the ILLIAC IV System and submit jobs for processing. Statements in the language consist of subsystem names which are normally accompanied by parameters supplied by the user (arguments). All ACL statements are structured according to fixed rules of syntax. The syntax of ACL statements described in this document is presented to a great extent through notational conventions. This section introduces the structure of the "basic" ACL statement. See Section 6 for a discussion of the "complete" ACL syntax.

3.1 GENERAL DESCRIPTION

ACL statements may be entered from a terminal in the interactive mode, with interaction between the user and the subsystem taking place on-line. A sequence of ACL statements may also be created as a file, and submitted as a batch processing job. Statements that call subsystems that require use of certain system resources, i.e., the B6700 or the ILLIAC IV, must be processed in batch mode. B6700 resources are required for the GLYPNIR compiler, the ASK assembler, and the SSK simulator. Once the user has prepared a batch job, an ACL statement (SUBMIT) can be issued in the interactive mode to place the job in the batch queue; other statements are used in the interactive mode to determine the status of a batch job. (The language is identical whether an ACL statement is entered interactively or in a batch job.)

3.2 ECHOING

Characters input by the user at his terminal are received by the ILLIAC IV System and typed out at the user terminal — a process referred to as "echoing." This echoing of interactive input gives the user a permanent record of his session and aids in the detection of transmission errors.

User passwords which are input at the terminal are not echoed, as a security precaution (except in the SUBMIT, COPY, and DELJOB statements).

3.3 CONVENTIONS USED IN THIS GUIDE

ACL statements must conform to certain rules which are indicated in the format representations in this guide. To make these rules clear to the user, a standard set of conventions is used throughout this guide.

```
EXAMPLE argument1,argument2{,argument3}ƀ
```

The conventions shown in this example of a format representation are:

- Words printed in all capitals must be entered literally, as illustrated by the word EXAMPLE above.
- Formal arguments to be replaced by the user with actual arguments are printed in lower-case letters (in the example above, "argument1,argument2", and so forth). An argument may be thought of as information for a subsystem. When formal arguments are printed in text (outside of the format representation), they are always underlined, e.g., argument1, and refer directly to the format representation.
- Braces are used as a notational convention to indicate that the enclosed item(s) are optional, but the braces themselves are not part of the statement, i.e., braces are not characters to be input to the system.
- All other punctuation marks, i.e., commas, periods, colons, semicolons, quotation marks, the equal sign, parentheses, and carriage returns, must appear exactly as shown. The symbol ƀ indicates a carriage return.

The following example of convention usage is taken from Section 7 of this guide.

```
LINKED {infilename},outfilename{,OPTIONS=optionstring}␢
```

where LINKED is a call to the link editor and must be entered literally.

- {infilename} is a name to be optionally supplied by the user; if not supplied, a default action or default name is assumed by the system as explained in the statement description.
- outfilename is a name that MUST be supplied by the user (because outfilename is NOT enclosed in braces). The comma must precede outfilename regardless of whether infilename is specified (because the comma is not enclosed in the braces). The comma in ACL is the delimiter which separates arguments and, when an argument is optionally omitted, the comma is required and functions as a placeholder.
- optionstring is a string of characters specifying the link editor processing options the user is selecting. OPTIONS= optionstring may be omitted, and if so, the final comma may also be omitted. The word OPTIONS must be entered literally if optionstring is to be specified, since it is in capitals in the format; likewise the equal sign (=) must be entered if optionstring is to be specified.

3.4 THE BASIC ACL STATEMENT FORMAT

The basic format for all ACL statements is as follows:

```
SUBSYSTEM-NAME {argument1,argument2,...,argumentn}␢
```

In the above simple statement format:

- SUBSYSTEM-NAME is the name of the desired ACL subsystem (such as DEL, DIR, or INQ) and must be input literally. SUBSYSTEM-NAME is used hereafter synonymously with the phrase "ACL statement name."

- arguments represent parametric information (such as the name of a file, a macro-call, or a value) supplied by the user and passed to the subsystem specified by the SUBSYSTEM-NAME. The rules for specifying actual arguments are discussed in paragraph 3.4.1 below.

The semicolon (;) performs one of two functions, depending on how it is used:

(1) When placed immediately after a comma that terminates an argument, or immediately following the blank(s) after the SUBSYSTEM-NAME, the semicolon is a continuation character; it may be followed by a carriage return and the statement continued on the next line.

(2) When used as the first character on a line that is not a continuation of the previous line, the semicolon is a comment character; it causes the entire line up to the carriage return to be interpreted as a comment. The comment may be any string of characters, terminated by the carriage return.

These two rules are not a complete or rigorous statement of the semicolon convention; see Section 6 for a complete statement. The following examples illustrate the simple use of the semicolon.

Example 1

```
DEL DATAFILE.A,DATAFILE.B,DATAFILE.C,;↵
DATAFILE.D,DATAFILE.E↵
RENAME INFILE.A,DATAFILE.A↵
```

The semicolon at the end of the first line is a continuation character; the second line is a continuation of the first, and the carriage return at the end of the second line terminates the first statement; a new statement appears on the third line.

Example 2

```
GLYP BIG.SOURCECODE,BIG.OBJECTCODE,BIG.LISTFILEb̸
;COMPILE BIGPROGRAM TO RELOCATABLE OBJECT CODEb̸
;AND PRODUCE A LISTINGb̸
DIR BIG.OBJECTCODE,BIG.LISTFILEb̸
```

The first line is a statement; the second and third lines are comment, and the fourth line is another statement.

3.4.1 ARGUMENT LIST RULES

The first argument in an ACL statement must be separated from the SUBSYSTEM-NAME by at least one space. Subsequent arguments must be separated by commas and must appear in the order shown in the format representations. Blanks may be inserted on either side of the comma. If an {optional} argument is omitted, its position must be accounted for by including the comma(s) that would normally separate it from adjacent arguments. For example, to omit infilename in the LINKED statement

```
LINKED {infilename},outfilename{,OPTIONS=optionstring}b̸
```

the correct form for an actual statement where the outfilename is GOFILE.A and the optionstring is XM would be

```
LINKED ,GOFILE.A,OPTIONS=XMb̸
```

If the omitted argument is the last one in the calling sequence, the comma after the next to the last argument may also be omitted. For example,

```
LINKED ,GOFILE.Ab̸
```

would be a correct construction with both of the optional arguments (infilename and OPTIONS=optionstring) omitted.

3.4.2 ARGUMENTS

An argument in an ACL statement is information passed to the subsystem called. Accordingly, the rules for the construction of arguments are generally defined by rules within the subsystems, and are not rules of ACL. Although some subsystems will accept special (non-alphanumeric) characters, others will not. The user is therefore cautioned against using special characters within arguments (e.g., AR;G1 or NEW-NAME) unless explicitly called for in the syntax of the argument (see Section 3.4.2.1 below).

There are several classes of arguments, and there is uniformity within the subsystems as to the construction of arguments within certain classes (for example, filenames have a standard syntax). Several classes are

(1) filenames
(2) expressions
(3) control parameters
(4) values
(5) keyword arguments

Each of the classes of arguments has rules governing its use and construction. Throughout this document, the word <u>argument</u> is avoided wherever possible and the class name to which the argument belongs is used in its place. This is done to give the reader a better understanding of each ACL statement, and to allow general rules of construction to be used throughout.

Note that not all arguments can be readily grouped into classes. Using the LINKED example above, <u>optionstring</u> is information required uniquely by the LINKED subsystem. Therefore, no attempt is made to describe it in general. For arguments not in general use, rules of construction, values, and usage conventions are included within the particular ACL statement descriptions (see Section 7).

An example of the use of argument class notation:

```
DEL filename↵        (delete the file
                      named filename)
```

Here, rather than using the general term argument, the format for this ACL statement is given with the argument class name.

Certain argument classes are used so extensively in ACL that general rules of construction may be given. This section includes only the rules for the construction of filenames. Filenames are the only argument class required to use the basic set of ACL presented in Section 4, so discussion of the other argument classes is deferred to Section 6.

3.4.2.1 Rules for the Construction of Filenames

A filename has the following format:

```
{<directoryname>}name{.extension}{;version}
```

where

- directoryname is the name of the directory with which the file is associated. In most cases, the file is associated with a user and the directoryname is the same as the userid. When a user omits directoryname from a filename, that user's userid is used as the default directoryname. Most of the ACL subsystems discussed in this guide deal only with files in the user's own directory; accordingly, it is normal to default the directoryname when specifying filenames in ACL statements. In the few cases where a directoryname other than the user's own userid may be specified, this fact is explicitly noted in this guide.
- name is any combination of up to 39 alphanumeric characters. Together with the extension (if any), it serves as the user's primary identifier for a file. If a set of files within the same directory have the same name and different extensions,

the name alone may be used in certain ACL statements to designate the entire set. If a filename has no extension, the name serves as the primary identifier for the file.

- extension is any combination of up to 39 alphanumeric characters. The extension is commonly used to differentiate among a set of files having the same name, or to serve as a descriptive mnemonic supplementing the name for the user's convenience. Most ACL subsystems that produce output files provide default extensions for the filenames. These default extensions are given in the descriptions in Section 7. In addition, certain subsystems provide default extensions for the filenames of input files. These are also given in Section 7.
- version is a sequence number assigned to a file to allow the user to create two or more files with the same name and extension. If the user omits the version (by leaving out the semicolon and number), the system selects a default value according to the following rules:

(1) If a new file is created (indicated by a new name or extension), version 1 is assigned.

(2) In reading a file from storage, the highest numbered version is selected.

(3) If changes are made to a file, and the modified file has the same name and extension as the original, the next highest version is assigned to the modified file.

(4) If a file is deleted without specifying a version, the lowest version is selected.

(5) If a file is renamed without specifying a version, the highest version is selected.

Caution: There is a limit to the number of versions of a file (with the same name and extension) that can be kept in a directory at the same time. The limit is not fixed, and the user will be kept informed of its current value. As new (higher-numbered) versions of a file are created, older (lower-numbered) versions will be automatically deleted to keep the total number of versions within the assigned limit. See Section 4.4.1 for further information.

3.4.2.2 Keyword Arguments

A keyword argument has the general form

```
KEYWORD=argument
```

The OPTIONS=optionstring argument in the LINKED statement shown above is an example; the KEYWORD in this case is OPTIONS. The KEYWORD and the "equal" sign must always be entered literally as shown in the syntax descriptions in this guide.

The following rules apply to keyword arguments:

(1) Keyword arguments are always optional;

(2) Any keyword argument that is valid in a given ACL statement may be inserted anywhere in the sequence of arguments for the statement, and separated from the other arguments by commas in the usual manner.

(3) When a keyword argument is omitted, one does not place commas in the statement to account for its position – since a keyword argument has no specific position in the statement syntax.

For example, the following LINKED statements are all equally valid and all mean exactly the same thing:

```
(1)  LINKED ,GOFILE.A,OPTIONS=XMb̸
(2)  LINKED ,OPTIONS=XM,GOFILE.Ab̸
(3)  LINKED OPTIONS=XM,,GOFILE.Ab̸
(4)  LINKED ,GOFILE.Ab̸
```

where commas are used as placeholders for the infilename argument, but are not needed as placeholders for OPTIONS=optionstring in (4).

In the syntax descriptions in Section 7, keyword arguments are shown in arbitrarily selected positions in the formats of ACL statements where they are valid, in cases where not many keyword arguments are valid. In cases where many keyword arguments are valid, they are listed separately.

3.5 SUBSYSTEM STATEMENTS

All ACL statements are calls to subsystems that process the arguments provided in the statement and perform the specified function. In addition to the arguments, certain subsystems require additional statements containing control information. These statements are called "control statements" or "subsystem statements." Interactively, subsystem statements are entered in response to a prompt, in most cases the colon (:) character.

There are two general formats for subsystem statements:

(1)

```
SUBSTATE argument1,argument2,...,argumentn↵
```

(2)

```
argument1:SUBSTATE argument2,argument3,...,argumentn↵
```

The subsystem statement sequence provided with the ACL statement is terminated by:

```
END↵
```

For example:

```
LINKED {infile},outfile{,OPTIONS=optionstring}↵
SET address↵
segname:SEG↵
INCLDE filename(s)↵
ENTRY label↵
END↵
```

is a complete set of ACL statements for a link editor subsystem call.

The notational conventions used in this guide for subsystem statements are the same as those previously described for ACL statements. Arguments in general follow the same rules for subsystem statements as for ACL statements. Each ACL subsystem described in Section 7 which has subsystem statements includes the format and usage descriptions of these statements.

NOTES: (1) The set of statements for DED, the text editor, is extensive. They do not conform to the simple rules above for construction of subsystem statements. Accordingly, a separate manual (Appendix A) is provided for the DED subsystem.

(2) The text string that is contained in a macro definition is <u>not</u> an ACL subsystem statement. This text is not restricted to any formatting rules (see MACRO, Section 7).

SECTION 4

TUTORIAL FOR THE BEGINNING ILLIAC IV SYSTEM USER

SECTION 4

TUTORIAL FOR THE BEGINNING ILLIAC IV SYSTEM USER

CONTENTS

SECTION 4

TUTORIAL FOR THE BEGINNING ILLIAC IV SYSTEM USER

4.1 INTRODUCTION

The following is a tutorial for the beginning ILLIAC IV System user. It includes a discussion of all the operations necessary to accomplish a basic ILLIAC IV process, beginning with entry to the System, proceeding through file preparation and file transfer to the submission of a job containing steps for compilation, assembly, link editing, layout of disk memory areas, data transfer to the ILLIAC IV memory system, program loading and execution, transfer of output data back from the ILLIAC IV, and finally the transfer of output data back to the user's Host and logout from the ILLIAC IV System.

The following section provides an introduction to the System's two basic modes of operation.

4.2 INTERACTIVE AND BATCH OPERATION

The ILLIAC IV System can be used in two modes, batch and interactive. In the interactive mode, the System accepts ACL statements from the user's terminal (via the ARPA Network) and responds to these statements by carrying out the operations they specify and by sending messages back to the user's terminal. <u>As long as you are logged in to the ILLIAC IV System from a terminal, you are using the System in interactive mode</u>.

In addition, it is possible to write a sequence of ACL statements in a file and place a request to process this file in the batch queue. While this request is in the queue, or while the file (which may be thought of as a deferred job) is being processed, the user may continue to operate interactively, or he may log out of the system. When the job is scheduled, the batch ACL file will be opened and ACL statements taken and processed in order, exactly as if they came from the user's terminal — except that system responses (messages, lists, etc.) are written out to a file instead of to the terminal.

A user who has submitted a batch job may still continue to work in interactive mode although he must be careful about referencing any files that are also referenced by statements in the batch job file.

The ACL statements in a file placed in the batch queue are scheduled and processed noninteractively. Certain statements — currently GLYP, ASK, SSK, and RUN — may be used only in a batch file.

The file of ACL statements to be processed in batch is called a primary input file or PIF. The file on which messages and other system responses are written out in batch is called primary output file or POF. The details of building an ACL statement file — structure of the PIF, the SUBMIT statement for entering a request into the batch queue, and other related statements — are discussed in Section 4.6 below, after a detailed discussion of the ACL statements required to set up a basic ILLIAC IV process.

4.3 ENTERING THE ILLIAC IV SYSTEM

It will be assumed here that you have been administratively assigned a userid and a password for entry to the ILLIAC IV System. This should be done through the ILLIAC User Liaison Representative for your organization.

The procedures for establishing a connection to the ILLIAC Host via the ARPA Network are not covered here, as they will vary from Host to Host. Once you are connected to the ILLIAC Host, the ILLIAC IV System will type a prompt character (@) on your terminal. In response to this, you must type a LOGIN statement, with the following format:

```
LOGIN userid password accountϸ
```

The userid and password are the ones administratively assigned to you. The password will not be echoed at your terminal.

The account may be any string of digits you wish to enter. It is not used by the system in the current implementation.

Note that the userid, password, and account are separated by spaces, not by commas as in the ACL statement format; LOGIN is not an ACL statement.

Once the LOGIN statement is completed, the system will respond with the ACL prompt character (!) and will be ready to accept an ACL statement.

4.3.1 THE HELP SUBSYSTEM

The HELP subsystem is provided to answer questions about the system typed in by the user. HELP can be called at any time in response to an ACL prompt character (!) typed by the system. To get HELP, type

$$\boxed{\text{HELP}\hookleftarrow}$$

HELP will respond by typing out a herald message followed by HELP's own prompt character, a question mark (?). In response to this prompt, type in a question terminated with either a carriage return or a question mark. HELP will type an answer followed by another question-mark prompt.

Anything typed by the user in response to a question-mark prompt is interpreted by HELP as a question; in order to enter an ACL statement, you must exit from HELP by typing

$$\boxed{\text{END}\hookleftarrow}$$

HELP will request an additional carriage return to confirm this, and upon receiving confirmation will terminate and the system will type an ACL prompt (!).

HELP operates by identifying keywords in the user's question and typing out stored answers associated with keywords or combinations of keywords. It cannot properly answer complex questions or questions requiring a YES or NO answer. Keep your questions as brief and simple as possible.

HELP's answers are written as succinctly as possible, and it may be necessary to ask a series of related questions in order to get the complete information desired on a given topic. When HELP cannot answer a question, it stores the question for examination by the project staff and requests the user to rephrase the question.

4.4 FILE TRANSFERS AND FILE MANAGEMENT

Very often the first thing you will want to do upon entering the system is to transfer files of information from your Host to the central file system. This system maintains a directory of all active files "owned" by each user — i.e., active files associated with the user's userid. This directory can be listed on your terminal by means of the DIR statement, which has the following format:

```
DIR {filename1,filename2,...,filenamen}↵
```

If the optional list of filenames is omitted from the statement format, the system will list the entire file directory; if one or more filenames are entered as part of the statement, only the directory entries for those files will be listed (thus providing a way to verify the information on a few particular files in the directory without listing the entire directory, which may be quite lengthy).

After verifying the status of your active files, new files may be transferred into the ILLIAC IV System by means of the CPYNET statement with the following format:

```
CPYNET (filename1,hostid,userid{,password{,account}}),filename2↵
```

where filename1 is the name of the file to be copied from your Host, hostid is the name or number of your Host, userid is the name of the user directory at your Host in which the file is held, password is the password associated with userid, account is a numeric string for user accounting purposes, and filename2 is the name of the destination file in the central file system. Password and account should be omitted if they are not required as part of the procedure for logging in to your Host.

Alternatively, you may create new files within the ILLIAC IV System by means of the text-editing subsystem, DED (described in Appendix A). DED is the most convenient means for creating relatively short files of alphanumeric text, such as a primary input file of ACL statements. Text files in the system may also be edited using DED.

4.4.1 FILE ARCHIVING

A file in the system may be either an active file or an archived file.

An active file is a file in the user's "active file storage space," whose name (with other information) appears in the user's file directory. Active files can be accessed directly by ACL subsystems, and throughout this guide, the term "file" refers to an active file unless otherwise noted.

An archived file is a file that has been copied from the user's active file storage space to the UNICON memory, and subsequently deleted from the user's directory. An archived file cannot be accessed directly by ACL subsystems, but must first be "restored," i.e., copied back into the user's active file storage space, as described below in Section 4.4.2.

The copying of files to the UNICON memory is done periodically during the day, without the need for any action by the user. Thus the user may archive an active file by taking the following steps:

(1) Verify that the file has been copied to the UNICON by using the UDIR statement (described below).

(2) Delete the file from active storage space and from the user's directory by using the DEL statement (described below).

(3) Save the printout produced on the terminal by UDIR.

The third step is important because in the present implementation, there is no way for the user to get a directory or listing of his archived files after they have been deleted from active status. UDIR provides complete information on a file that has been copied to the UNICON, but only while the file remains in the user's directory, i.e., before the archiving process has been completed by deleting the file from active storage.

The UDIR statement has the following format:

$$\boxed{\text{UDIR}\not{b}}$$

After this statement is entered, the UDIR subsystem will prompt you by typing "OUTPUT TO:". Respond by typing in a carriage return if you want the information typed out on your terminal, or with a filename followed by a carriage return if you want the information written to a file. (Note: When UDIR writes out information to an existing file, any information already in the file is overwritten and lost.)

UDIR will now prompt again by typing out "FILE NAME:". Again you may respond either with a carriage return or with a filename followed by a carriage return. In the first case, UDIR will respond by supplying information on all active files in your directory that have been copied to the UNICON; in the second case, it will return information on the specified file only.

The information returned by UDIR includes the filename, the number of pages in the file, the date and time when the file was copied to the UNICON, and the UNICON address of the file.

To complete the archiving process by deleting a file from your active file storage space, use the DEL statement with the following format:

```
DEL filename␢
```

where filename designates the file to be deleted. If this file has not been copied to the UNICON, it will be irrevocably lost.

The amount of active file storage space assigned to each user is limited to a certain number of pages, while the amount of space available for archived files may be regarded as virtually unlimited, since the UNICON memory has an extremely large on-line capacity and an unlimited off-line capacity.

To find out how much space is assigned to you for active file storage, use the MAXAFS statement, with the following format:

```
MAXAFS␢
```

The MAXAFS subsystem will respond by typing out the number of pages of active file storage space assigned to you.

If your active files exceed the assigned number of pages, one or more of your active files may be archived for you at the end of a working day — i.e., one or more files that have been copied to the UNICON may be deleted from your active file storage space and from your file directory without your intervention.

4.4.2 RESTORING ARCHIVED FILES

To restore archived files, use a NOTIFY statement as described below to send a message. The message should include the following information:

(1) filename including at least the directoryname, name, and extension and preferably the version;

(2) Archiving information. If you have UDIR information for the file, give the UNICON address and the date and time returned

by UDIR. Otherwise, give a pair of dates (and possibly times) between which the file was copied to the UNICON, or the last date and time when you know that the file was in your directory.

The more exact the archiving information given in the message, the faster the file can be restored. If UDIR information is routinely obtained and saved by the user before files are archived, restoration will normally be a quick process.

To send a message, use the NOTIFY statement with the following format:

```
NOTIFY "message"ɀ
```

where the message may be of any length and may include any character except quotation marks; in particular it may contain carriage returns. The quotation marks delimiting the message may not be omitted. The following example illustrates the suggested form for the message:

Example:

```
NOTIFY "RESTORE <JONES>DATA.BATCH;23 (JUNE 25 73,JUNE 30 73).ɀ
RESTORE <JONES>PROG.SUBRS;2 (JUL 11 73 15:04 00021,16571,5)."ɀ
```

Here the user is requesting that two files (<JONES>DATA.BATCH;23 and <JONES>PROG.SUBRS;2) be restored to active status. For the first file, the user does not have UDIR information, so he supplies a pair of dates between which the file was copied to the UNICON. For the second file, he gives the date, time, and UNICON address obtained from UDIR while the file was still in his directory.

A staff member should acknowledge the message within a few minutes by linking to the user's terminal and typing a message. If no acknowledgment is received within a few minutes, use the NOTIFY statement again to repeat the message.

Note that when archived files are required for the processing of a batch job, you are not required to request explicitly that they be restored; this will be done as matter of routine.

4.5 BASIC OPERATIONS AND ACL STATEMENTS

For purposes of this discussion, we will assume that your objective is to start with one or more files of GLYPNIR source code and one or more files of data, compile, assemble, and link-edit the code, transfer it and the data to the ILLIAC IV or input it to the ILLIAC IV Simulator (SSK), and ultimately transfer the output data (if any) back to your Host. We will now examine this sequence in a little more detail with respect to the ACL statements required.

4.5.1 COMPILATION

The first step is compilation and assembly of GLYPNIR source code. This is done by means of the GLYP statement, with the following format:

```
GLYP infilename,outfilename{,listfilename}␢
```

where infilename is the filename of a file of GLYPNIR source code, outfilename is the filename of the file to contain the relocatable ASK object code, and listfilename is the name of the file to contain the listing produced by the compiler. The outfile and the listfile need not already exist. If listfilename or outfilename includes a version number and that version number exists, the contents will be overwritten; if the files do not exist, new files will be automatically created. Furthermore, the listfilename may optionally be omitted from the statement in which case no listing will be produced. The various options of the GLYPNIR compiler are controlled by statements included in the source code file.

In the current implementation, GLYP is illegal in interactive mode; it must be used in a batch file.

4.5.2 LINK EDITING

The compilation and assembly process produces relocatable object code, which must be converted to an absolute-address ILLIAC IV SAVE file (ISV file) for execution by the ILLIAC IV or SSK. This is done by the link editor, LINKED. Input to LINKED is in the form of the LINKED statement followed by a sequence of control statements which specify the file(s) of ASK relocatable code to be included in the ISV file. The LINKED statement has the following general format:

```
LINKED {infilename},outfilename{,OPTIONS=optionstring}␢
```

where infilename is the name of a file containing LINKED subsystem statements to control the link editor, outfilename is the name of the output ISV file to be produced by LINKED, and optionstring is a string of letters specifying various optional features of LINKED.

The infilename may be omitted and the link-editor control statements entered in-line after the LINKED statement; for simplicity, we will follow this practice for purposes of this discussion. The OPTIONS=optionstring argument may also be omitted, and we will omit it throughout this discussion.

The LINKED control statements used in this discussion are INCLDE and END. There are two others called SET and ENTRY, which will not be discussed here.

The INCLDE statement has the following format:

```
INCLDE filename1,filename2,...,filenamenƀ
```

where the filename(s) are the names of files of relocatable ASK object code to be included in the ISV file.

The END statement serves to terminate the sequence of control statements to LINKED, and is mandatory. It has the format

```
ENDƀ
```

After the END statement, LINKED will perform the link editing, produce a page map of the output file on the user's terminal or primary output file, and terminate. The system will then type an exclamation-point (!) prompt (in interactive mode) to show that it is ready for a new ACL statement. The file(s) specified in the INCLDE statement are collected into an ISV file ready for execution on the ILLIAC IV or simulation by SSK. (The ISV file contains all the necessary information to specify the initial machine state of the ILLIAC IV.)

4.5.3 MAPPING DATA AREAS IN THE ILLIAC IV DISK MEMORY (I4DM)

Usually data structures to be processed on the ILLIAC IV will be much too large to load integrally into the working storage (Array Memory) of the ILLIAC IV Processor. Accordingly, a large dedicated disk memory system (the I4DM) functions as the main memory for the ILLIAC IV. Data and programs are transferred to this memory system and then transferred in and out of working storage.

The time required for I4DM data accessing will become great compared to the computation speed of the ILLIAC IV unless the I4DM data layout is correctly specified; this layout of data areas on the I4DM is critical to the efficient use of the ILLIAC IV Processor and must be specified in detail by the user.

The I4DM layout is specified by means of a MAP subsystem call. The MAP subsystem, like the link editor, requires control statements to specify its operation. As the explanation of these control statements is lengthy, it will not be discussed here; complete details are to be found in Section 5 and Section 7. The MAP statement itself has the following format:

```
MAP {infilename}{,mapfilename}ƀ
```

where infilename is the name of a file containing control statements for the MAP subsystem and mapfilename is the name of the file to contain the I4DM allocation tables generated by MAP.

As with LINKED, the infilename may be omitted and the control statements entered in-line after the MAP statement. In the remainder of this discussion, however, we will assume that the control statements are in an infile.

If mapfilename is omitted, no allocation tables are produced. A time or map listing can be output on the user's terminal (or in his primary output file in batch); this option may be used to analyze a tentative I4DM layout.

The effect of the MAP statement (and its file of control statements) is to create a file of allocation tables. No actual assignment of disk memory is performed; this is done with the ALLOC statement (see below).

The specification of an I4DM layout with MAP is always required when a program is to be run on the ILLIAC IV; however, if the program is merely to be simulated with SSK, I4DM mapping may be omitted.

4.5.4 I4DM AREA ALLOCATION

Once the layout of areas in I4DM has been defined by means of the MAP subsystem, we need to request the assignment of physical space on the disk. This is done by means of the ALLOC statement, with the format

```
ALLOC mapfilename{,allocid}b̸
```

where mapfilename is the name of the allocation table file to be produced by the MAP subsystem. The allocid may be omitted and will not be discussed here.

When the ALLOC statement is processed (immediately prior to loading the I4DM and executing an ILLIAC IV program), the system will examine the current allocation of I4DM space and will, if possible, assign the necessary space while other job(s) still have reserved space in I4DM. If it is impossible to assign all the necessary space, the allocation will fail. Note that the ALLOC statement does not cause any information to be transferred to the I4DM; it merely assigns the necessary space for data areas to the user.

The ALLOC statement is omitted if the user merely wishes to simulate a program with SSK.

4.5.5 TRANSFERRING DATA TO I4DM

After I4DM space is assigned according to the MAP specification, data is transferred by means of the MOVE statement. The procedure is to transfer from a file named in the user's directory to an assigned disk area. The format for the MOVE statement in this case is

```
MOVE filename,I4DM:areanameb̸
```

where filename is the name of a data file in the central file system and areaname is the name of an allocated I4DM area (previously defined and named by means of a control statement to the MAP subsystem). The characters "I4DM:" prefixed to the areaname are mandatory. Like the MAP and ALLOC statements, this MOVE statement is omitted if the program is to be simulated with ASK rather than run on the ILLIAC IV.

4.5.6 LOADING AND EXECUTING A PROGRAM ON THE ILLIAC IV

When all the steps described above have been performed, the system is ready to load and execute an ILLIAC IV program. This is done by means of the RUN statement, with the following format (for purposes of this tutorial):

```
RUN filenameƀ
```

where filename is the name of an ILLIAC IV SAVE (ISV) file produced by the LINKED subsystem.

When the RUN statement is processed, the ISV file is transferred to the portion of I4DM reserved for programs; is is then loaded into Array Memory and execution is started with a transfer of ILLIAC IV control to the initial entry point in the user's program.

In the current implementation, RUN is illegal in interactive mode; it must be used in a batch file.

4.5.7 TRANSFERRING DATA FROM I4DM

Normally, the first thing to do after program execution on the ILLIAC IV is to transfer the output data from the ILLIAC IV Disk Memory back to a file or files in the user's directory. This is done by the exact reverse of the procedure for putting data on the disk, namely a MOVE statement following the RUN statement. This MOVE statement transfers data from a disk area to a file. The following format is used:

```
MOVE I4DM:areaname,filenameƀ
```

where areaname is the name of a disk area (defined by a control statement to the MAP subsystem) containing output data and filename is the name of a file to contain that data. The characters "I4DM:" prefixed to the areaname are mandatory.

This step is not necessary after a program has been simulated by SSK, as SSK will output data directly to a file in the user's directory.

4.5.8 TRANSFERRING DATA FILES FROM THE ILLIAC HOST TO THE USER'S HOST

The final step is to transfer the file or files of output data back to the user's Host. Once again, the CPYNET statement is used, with the following format:

```
CPYNET filename1,(filename2,hostid,userid{,password{,account}})↵
```

where filename1 is the name of a file held in your directory in the central file system, filename2 is the name of the file at your own Host to which you wish to copy the contents of filename1, hostid is the name or number of your Host, userid is your user identification at your Host, password is the password associated with your userid at your Host, and account is a numeric string for user accounting purposes. Password and account should be omitted if they are not required for logging in to your Host.

4.6 BATCH REQUESTS: THE PRIMARY INPUT FILE (PIF)

We have just gone through a discussion of the basic ACL statements used in preparing and running an ILLIAC IV program, without considering the means for entering these statements into the system for processing. As discussed earlier, most ACL statements can be entered directly from the user's terminal in interactive mode, but some — the GLYP, ASK, SSK, and RUN statements — are currently illegal in interactive mode and must be submitted in a batch job. A batch job is the processing of ACL statements contained in a file and submitted to the batch-processing facility of the system by means of the ACL statement SUBMIT.

Such a file is called a primary input file or PIF. As a rule, a user will either bring a PIF into the central file system from his own Host (using the CPYNET statement) or construct it interactively using the ILLIAC IV System's text editor, DED. He may then submit a request to process it with a SUBMIT statement and log out, since the PIF is processed noninteractively.

4.6.1 STRUCTURE OF A PRIMARY INPUT FILE

A PIF may contain any legal sequence of ACL statements; here we will discuss what is essential in order to have a meaningful and useful PIF, using the same processing sequence discussed above. We assume that the source code is written in GLYPNIR source language and that program and data files originate outside the ILLIAC IV System and must be brought in by means of CPYNET statements.

As a minimum, the PIF must contain the GLYP and RUN statements, since currently these may be used only in batch. Input data should be transferred to the I4DM just prior to program execution on the ILLIAC IV, so the RUN statement should be preceded by one or more MOVE statements for this purpose; and the MOVE statements must be preceded by an ALLOC statement to assign I4DM space. Thus we have the sequence GLYP, ALLOC, MOVE, RUN. Additionally, the PIF must contain any other statements that are required between the GLYP and RUN statements; for purposes of this discussion, these are the LINKED statement and its associated control statements.

At the end of the batch job run, all I4DM space assigned in the job is marked for deallocation, and data remaining in I4DM may be lost. Therefore, the PIF should also contain one or more MOVE statements following the RUN statement to transfer ILLIAC IV output data from I4DM to files in the user's directory.

Thus a typical PIF might contain the following broad structure:

(1) One or more GLYP statements to compile and assemble GLYPNIR source code;

(2) A LINKED statement (with control statements following in-line or else in another file) to create an ISV file from the assembled code;

(3) An ALLOC statement to allocate the I4DM areas defined and named by control statements to the MAP subsystem;

(4) One or more MOVE statements to transfer information to I4DM;

(5) A RUN statement to load and execute the program (using the ISV file produced by the LINKED subsystem);

(6) One or more MOVE statements to transfer data from I4DM to files in the user's directory.

4.6.2 THE CREATION OF A PIF

The PIF can be created at the user's Host and then transferred to the ILLIAC IV System by means of a CPYNET statement used interactively (assuming that it is in a form compatible with the ILLIAC IV System), or it can be created by interactive use of the DED text editor. The text editor is a subsystem called by the ACL statement DED, which has the format

```
DEDb̸
```

DED is controlled by a large set of DED command statements, which are discussed in detail in Appendix A and will not be discussed further in this section.

The formats of statements written in the PIF are exactly the same as if they were being entered from the user's terminal. The character (!) may be used as the first character of any ACL statement in a PIF, but the system ignores it and it has no meaning.

Since the system will respond to statements in a PIF exactly as if they were being entered from a terminal, it must have a place other than the terminal to write out messages, certain types of listings, etc. A file in the user's directory is used for this purpose and is called the primary output file or POF.

4.6.3 EXAMPLE OF PRIMARY INPUT FILE

We will now give an example of a sequence of ACL statements that would constitute a functionally complete primary input file for a simple batch job. Functional groups of statements are followed by commentary. It is assumed that the user's file directory in the ILLIAC IV System contains the following files: SOURCE.ONE and SOURCE.TWO, which are two files of GLYPNIR source code; DATAFILE.IN, which is a file of input data; and DISKMAP.FOO, which is a file of allocation tables created by a prior call to the MAP subsystem (see Section 5). In this MAP call, the user has declared two I4DM data areas named INAREA and OUTARA.

```
GLYP SOURCE.ONE,ASKCODE.ONE,GLYPLIST.ONEb̸
GLYP SOURCE.TWO,ASKCODE.TWO,GLYPLIST.TWOb̸
```

These two ACL statements cause the GLYPNIR source-code files, SOURCE.ONE and SOURCE.TWO, to be compiled and assembled one after the other. The output is relocatable ASK object code, written in files ASKCODE.ONE and ASKCODE.TWO, and the compiler listings are written in files GLYPLIST.ONE and GLYPLIST.TWO.

```
LINKED ,GOFILE.FOOb̸
INCLDE ASKCODE.ONE,ASKCODE.TWOb̸
ENDb̸
```

This LINKED statement omits an infilename (a comma is used as a placeholder for it), and so the control statements are in-line. The INCLDE statement specifies files ASKCODE.ONE and ASKCODE.TWO as the ASK object code for the ISV file. The output ISV file will be named GOFILE.FOO.

```
ALLOC DISKMAP.FOOb̸
```

This statement causes space in I4DM to be assigned for the areas specified in file DISKMAP.FOO.

```
MOVE DATAFILE.IN,I4DM:INAREAb̸
```

This causes the contents of file DATAFILE.IN to be transferred into I4DM area INAREA.

```
RUN GOFILE.FOOb̸
```

The ISV file GOFILE.FOO is placed in I4DM and then loaded into working storage. Execution will begin at the initial entry point. The program will contain calls for transferring data between working storage and the data areas in I4DM (INAREA and OUTARA). At program termination, the output data will be in OUTARA.

```
MOVE I4DM:OUTARA,DATAFILE.OUTb̸
```

This transfers the contents of area OUTARA into the central file system file DATAFILE.OUT.

NOTE: Ordinarily a user will want to verify the success of his job before initiating large data file transfers back across the ARPA Network. Assuming the job is successful, the following sequence of CPYNET statements will transfer data back to the user's Host. It is assumed here that these CPYNET statements are entered interactively.

```
CPYNET DATAFILE.OUT,(OUTDATA.GOODSTUFF,MOON,FANSOME,XYZ,102)ƀ
CPYNET GLYPLIST.ONE,(GLIST.ONE,MOON,FANSOME,XYZ,102)ƀ
CPYNET GLYPLIST.TWO,(GLIST.TWO,MOON,FANSOME,XYZ,102)ƀ
CPYNET FANPOF.FOO,(DIAG.FOO,MOON,FANSOME,XYZ,102)ƀ
```

These statements copy files from user Fansome's directory in the central file system to his directory at his own Host (the Moon Host). Fansome's <u>password</u> is XYZ. The <u>account</u> number is 102. The first statement copies his output data; the second and third copy the listing files produced by the GLYPNIR compiler; and the last statement copies the primary output file, FANPOF.FOO. This file has not yet been created at the time the primary input file is written; it is specified when the PIF is submitted for batch processing with a SUBMIT statement.

4.6.4 THE SUBMIT STATEMENT

Once a suitable PIF exists in the user's directory, he may submit it for batch processing with the SUBMIT statement, which has the following format:

```
SUBMIT pifname,{pofname},{runcode},userid,password,accountƀ
```

where <u>pifname</u> is the name of your PIF, <u>pofname</u> is the name to be assigned to your POF, <u>userid</u> is your user-identification code, <u>password</u> is your password, <u>account</u> is a numeric string, and <u>runcode</u> is one of the following:

0 if your PIF contains no statements requiring either the B6700 or the ILLIAC IV resources;

1 if your PIF requires the ILLIAC IV resource (i.e., if the PIF contains a RUN statement or any MOVE statements);

2 if your PIF requires the B6700 resource (i.e., if the PIF includes any GLYP, ASK, or SSK statements);
3 if your PIF requires both the ILLIAC IV resource and the B6700 resource.

If the runcode is omitted, the default value is 3.

If the pofname is omitted, the primary output file will have a filename made up of the pifname as the name and POF as the extension.

After the SUBMIT statement is completed, the system will type a number called the jobid. This number serves to identify the job in INQ or DELJOB statements (discussed below). The system then types a prompt character (!) and awaits a new ACL statement.

You may now log out, as the batch job requires no further intervention; or you may remain logged in, entering further ACL statements in interactive mode (possibly submitting another batch job, for example). If you remain logged in, you should be careful not to alter or delete any files referenced by statements in the submitted PIF.

4.6.5 THE INQ AND DELJOB STATEMENTS

The INQ statement is a means for finding out the status of a batch job after it has been submitted. The format of INQ is (for purposes of this tutorial):

$$\boxed{\texttt{INQ jobid}\hookleftarrow}$$

where jobid is the number typed by the system in response to the SUBMIT statement. The system responds to the INQ statement with one of the following messages:

WAITING	The job is still awaiting its turn for processing.
RUNNING	The job is being processed.
COMPLETED	The job is neither waiting nor running. (Note: This message may be misleading if a considerable time has elapsed since the job was submitted.)

A batch job that has been submitted with a SUBMIT statement may be "unsubmitted" with the DELJOB statement. The DELJOB statement has the following format:

```
DELJOB jobid,userid,password␢
```

where jobid is the number typed by the system in response to the SUBMIT statement, userid is the userid entered in the SUBMIT statement, and password is the password associated with that userid. The job will be deleted from the queue. If the job is being actively processed, the DELJOB statement has no effect.

SECTION 5

DATA TRANSFERS AND THE MAP SUBSYSTEM

SECTION 5

DATA TRANSFERS AND THE MAP SUBSYSTEM

CONTENTS

SECTION 5

DATA TRANSFERS AND THE MAP SUBSYSTEM

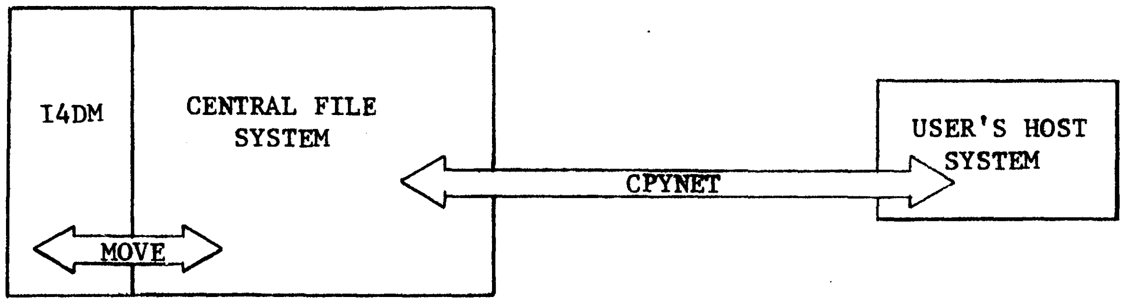
This section includes necessary information and a set of basic procedures for transferring data between a user's Host system and the ILLIAC IV Processor. The procedures described here are very general and are sufficiently detailed only for a limited set of ILLIAC IV users. Users with more complex data problems will find this discussion useful in providing a basic understanding of the ACL subsystems supporting data transfers to and from the ILLIAC IV Disk Memory (I4DM).

This section includes an extended discussion of the ACL subsystems MAP and ALLOC which perform I4DM preparation tasks for the ILLIAC IV user. These two subsystems must be called by ACL statements at some time prior to data transfers to the I4DM. Since the I4DM functions in the system as the main memory store for the ILLIAC IV Processor, every ILLIAC IV user must acquire a working knowledge of these necessary preparation tasks.

5.1 DATA TRANSFERS — AN OVERVIEW

In this introductory discussion, data movement from its source to the ILLIAC IV is viewed as having three nodes: a user's own local computing system (user Host), the central file system, and the I4DM. Data transfer is a two-step sequence using a CPYNET statement and a MOVE statement, as seen in Section 4. The first step is the transfer of data from the user's Host to the central file system. The second step is the transfer of data from the central file system to the I4DM. Data movement from the I4DM back to the originating source is the inverse of this sequence.

The following diagram illustrates the transfers. The arrows in the diagram represent transfers performed by the MOVE and CPYNET subsystems.



The data transfer rate across the ARPA Network is something less than 50K bits/second, while the data transfer rate of the I4DM is approximately 10^9 bits/second. One of the functions of the central file system in data transfers is to buffer the data flow from the user's Host to the ILLIAC IV so as to smooth this difference in transfer rates.

The request to transfer data from the user's Host to the central file system is shown in the following format:

```
CPYNET (filename1,hostid,userid{,password{,account}}),filename2ɓ
```

See Sections 4 and 7 for detailed explanations of this format.

The request to move data from the central file system to the I4DM is of the following form:

```
MOVE filename2,I4DM:areanameɓ
```

In these two calls, filename2 is the name of a file residing in the central file system. In moving data from a source to the ILLIAC IV, the user need not be concerned with the system's management of this file — which may involve, for example, staging on various storage devices in the file system. A user file in the central file system may be accessed, edited, moved, or deleted by the user through ACL, without concern for where it is located in the file system or how it is managed by the operating system software.

5.1.1 DATA TRANSFERS BETWEEN THE USER AND THE CENTRAL FILE SYSTEM

Data transfers discussed in this section are presumed to be for the purpose of bringing input data to the ILLIAC IV, processing it there, and bringing results back to the user. Therefore, this input data must ultimately conform to the formatting rules of the ILLIAC IV Processor.

- Data Transfer Protocol — The central file system supports the ARPA Network File Transfer Protocol, FTP.
- Data Word Formats — It is expected that ILLIAC IV user source data will originate (or be collected) in a variety of formats. The CPYNET subsystem logically performs, as its basic option, a serial bit string transfer between the user Host and the central file system. Data prepared by the user and transferred with the CPYNET subsystem, without conversion, will subsequently be written to the I4DM (via a MOVE call) in the bit sequence of the source file. It is possible, therefore, for the user to originate or convert data to ILLIAC IV formats external to the central file system, e.g., within the user's Host environment, and to transfer the data to the I4DM without further data format manipulation.

Alternately, source data which is not in ILLIAC IV word formats may be transferred to the central file system and converted there for subsequent use by the ILLIAC IV Processor. To this end, extensive data word format conversion programs are being prepared and placed in the system for general use.

User information for these subsystems will be provided as soon as the programs are available.

Finally, a special conversion program called CONVRT is available for use with the ILLIAC IV Simulator, SSK. See SSK in Section 7 of this guide.

- Data File Formats — The CPYNET subsystem transfers files in strict page sequence. No structure or key information is assumed to exist within the file pages; if it does, it will be treated as data.

5.1.2 DATA TRANSFERS BETWEEN THE CENTRAL FILE SYSTEM AND THE ILLIAC IV

Once data has been placed in the central file system, it may be transferred to the I4DM. Similarly, output data from ILLIAC IV processes must be copied from the I4DM back to the central file system.

However, prior to a data transfer with the I4DM, preparation in the form of a disk layout and space assignment must be accomplished by the user. The ACL subsystems supporting these preparation tasks are MAP and ALLOC. The nature of these preparations will be examined in Section 5.3 and the following.

5.1.3 SUMMARY OF DATA TRANSFER OVERVIEW

The model of data movement presented here is introductory only. To summarize the process, the user prepares source data external to the ILLIAC IV System either in the word format required by the ILLIAC IV Processor or in a format that is converted using ILLIAC IV System resources. Data is transferred first to the central file system, where it is collected and staged (and optionally converted), then to the I4DM; inversely, data is transferred from I4DM to the central file system, and from there to the user's Host. All transfers are performed by the MOVE and CPYNET subsystems.

The data rate smoothing functions performed by the central file system should be transparent to the user. However, the user must be concerned in greater detail with the I4DM preparation requirements, and with data transfers within the ILLIAC IV (i.e., between I4DM and Array Memory). Extensive control of these transfers is critically important to the ILLIAC IV user. The remainder of Section 5 is principally concerned with I4DM preparation.

5.2 THE ILLIAC IV MEMORY SYSTEM

Data transfers to the ILLIAC IV are accomplished using the COPY subsystem, which makes the I4DM directly accessible to the user through ACL. The I4DM is an integral part of the ILLIAC IV memory structure, and as such it requires certain preparations before the user may transfer data to it. The reasons become evident on closer examination of the ILLIAC IV memory hierarchy.

5.2.1 THE ILLIAC IV MEMORY HIERARCHY

Section 2 of this guide presents an overview of the ILLIAC IV System; the reader should be familiar with this overview. This section describes the ILLIAC IV Memory System in greater detail.

The objective of data transfers to the ILLIAC IV is to provide data to the ILLIAC IV Processor. This processor is an array of 64 individual processing elements (PE's). Only data resident in the working storage of the array may be immediately operated on by the PE's. This working storage is the Array Memory, which has a capacity of 128K (64-bit) or 256K (32-bit) words and a cycle time of 313 nanoseconds. The Array Memory is also used, incidentally, for storage of the currently executing program segment(s).

It is expected that ILLIAC IV processes will require large volumes of data (compared to PE working storage). The main memory store for the PE array is the I4DM.

5.2.2 THE ILLIAC IV DISK MEMORY (I4DM) SYSTEM

The following sections describe the disk hardware characteristics and the "logical disk" presented to the user by the ILLIAC IV I4DM management software. The hardware is controlled by system utility software and the user does not interact directly with the physical disk structure. All user interaction with the disk memory system is via utility software, which presents a convenient logical model of the disk structure. Accordingly, the following discussion of the physical disk system is brief, while the discussion of the logical disk is more extensive and detailed.

5.2.2.1 The Physical Disk Memory System

Physically, the I4DM is a fixed-head rotating disk system with a capacity of approximately 16×10^6 (64-bit) words or 32×10^6 (32-bit) words. The system is composed of 13 disks attached to two controllers (six disks on one controller, seven on the other). The disks rotate synchronously with a 40-millisecond rotation period. The maximum data-transfer rate is about 10^9 bits/second.

5.2.2.2 The Logical Disk

The software for management of the I4DM presents to the user a logical model of the disk system, called the "logical disk." The characteristics of this model are derived from the physical disk system, and are described in detail here. This description is presented from a logical perspective; that is, the way the system is to be viewed by a user. It is not necessary to relate this logical structure to physical characteristics of the disks.

The logical disk system is divided into 52 bands. There are four bands on each of the 13 disks, arranged concentrically (see Figure 5-1).

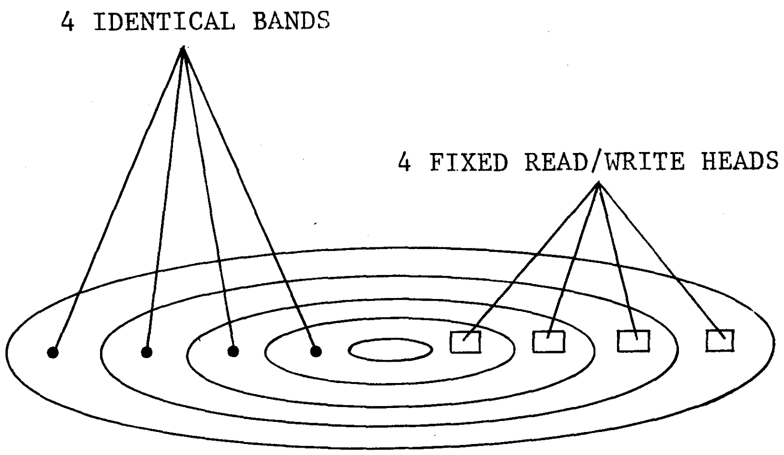


Figure 5-1. A Logical Representation of One of the 13 Identical Disk Units of the I4DM

All bands are identically formatted. Each band has its own read/write head, and logically only one head may be activated at one time. There is a time delay (for electronic switching) in accessing from one band to another (i.e., switching heads), and in switching between the read and write modes. In conjunction with the implementation of the logical disk, these delays have been generalized to the following rule. There is a time delay equal to 266 microseconds or two pages of rotation time (see below)

(1) between any two successive disk access requests (e.g., read-read, read-write, etc.);
(2) for each band switch that occurs.

Since every band is formatted identically to every other band, and since the access time from any band to any other is a constant, there is no logical contiguity among the bands. That is, no band is closer to a given band than any other band, in terms of access time. The 52 bands are numbered 0-51 for identification purposes.

Each band is divided into 300 pages. A page contains 1024 (64-bit) or 2048 (32-bit) contiguous words. A page is the smallest addressable unit of information on the disk, and thus the subsequent discussion of the disk will be principally in terms of pages and sets of pages. The 300 pages on a band are contiguous, and are numbered in sequence from 0 to 299 in the direction of rotation of the physical disks. Within a band, any given page (say, Page n) is contiguous to the pages before and after it in the numbering sequence (Pages n-1 and n+1). Page 0 and Page 299 are contiguous, i.e., Page 299 is followed by Page 0 as the disk rotates.

This attribute of contiguity for sequential pages within a band means that the rotational delay to access a set of contiguous pages is equal only to the rotational delay to access the first page of the set. An access of a set of noncontiguous pages incurs some rotational delay between pages of the set.

Since the disks are synchronized, the pages passing the read/write heads for all bands at any given moment all have the same page number.

Each band can be conveniently visualized as a circular strip of 300 pages, as shown in Figure 5-2. This drawing also illustrates the fact that the bands are all synchronized. Note that the bands are shown as identical without a numerical sequence; i.e., the band numbers are omitted from the illustration to emphasize the fact that they are strictly notational.

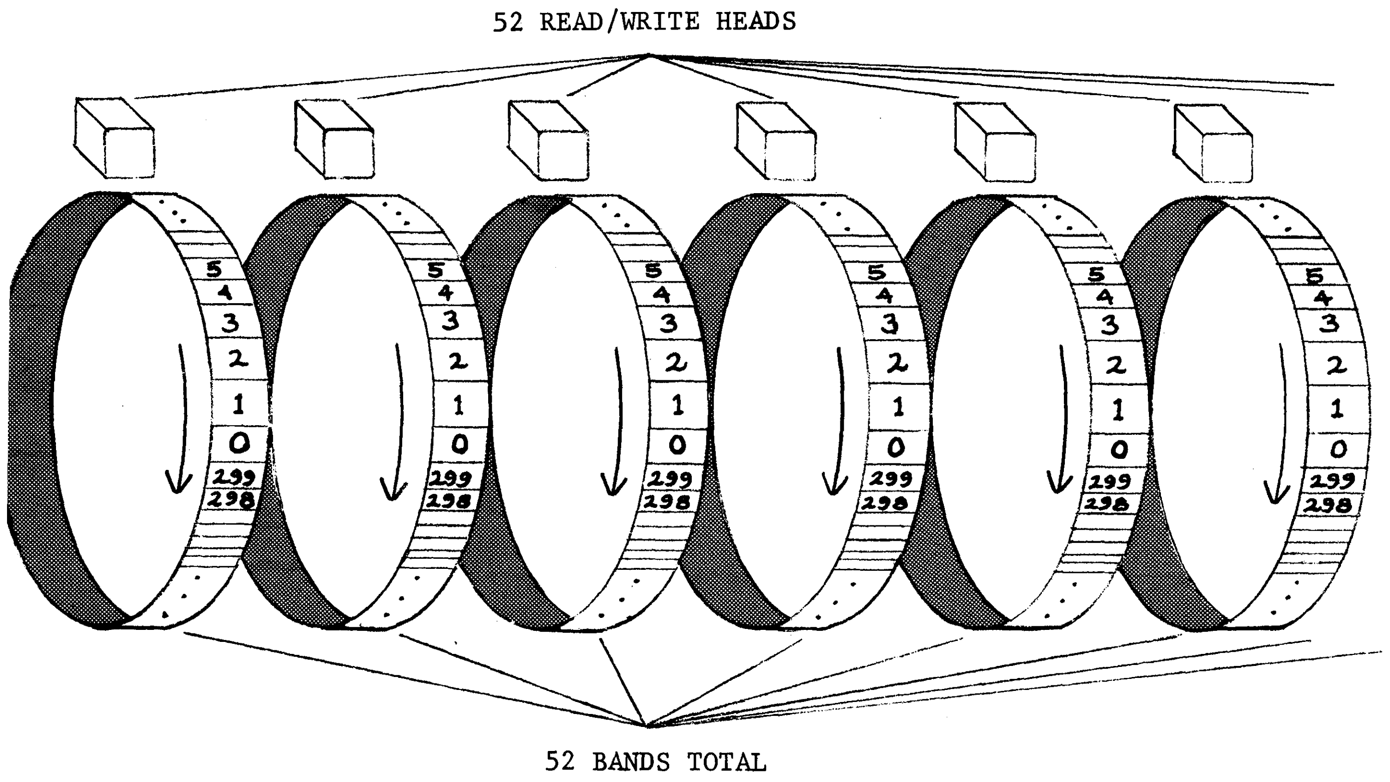


Figure 5-2. Logical Disk Structure, Showing Six of the 52 Bands

Given that the rotational period of the disk is 40 milliseconds, and that there are 300 pages on a band, each page is under the read/write heads for 133 microseconds. This is the transfer time for a page of data. Also, the 266-microsecond switching intervals mentioned above are thus equal to the time required for two pages to pass the read/write heads.

Summary of the Logical Disk

- There are 52 identical, synchronized bands, with a constant (266 microsecond) switching time from any band to any other band.
- Each band has its own head, and logically only one head is activated at a time; i.e., only one band may be accessed at a time.
- The delay time between any two successive access requests is equal to 266 microseconds.
- Each band is divided into 300 contiguous pages, numbered 0 to 299, with Page 299 contiguous to Page 0.
- Each page = 1024 (64-bit) words or 2048 (32-bit) words. A page is the smallest addressable unit in I4DM.
- Access time for a contiguous set of pages is equal to the access time of the first page in the set.
- The transfer time for any single page is 133 microseconds.
- Switching time from one band to any other band = delay time between successive access requests = 266 microseconds = the rotational time for two pages to pass the heads.
- Because disks are synchronized, the bands are synchronized, and thus pages with the same number in different bands pass under the read/write heads at the same time.

5.3 ILLIAC IV DISK MAPPING: THE MAP SUBSYSTEM

Two ACL subsystems must be called prior to transferring data to the I4DM; these are MAP and ALLOC.

The MAP subsystem accepts a user-prepared I4DM layout description and produces a file of allocation tables for input to the ALLOC subsystem. The ALLOC subsystem takes the MAP allocation tables as input. ALLOC assigns the required I4DM pages relatively positioned as described in the user's layout.

The MAP subsystem may be run at any time prior to COPY from a file to I4DM. The ALLOC subsystem should be called immediately prior to the COPY. Upon successful completion of ALLOC, the user may call the COPY subsystem to transfer data to the I4DM.

The MAP subsystem call consists of a MAP statement and a series of control statements. These statements may be contained in a separate file or they may be in-line following the MAP statement. The control statements are FORMAT, PRINT, TIME, and END. They are described in detail below. The MAP statement itself has the following format:

```
MAP {infilename}{,mapfilename}b̸
```

where

- infilename is the name of a file containing control statements for input to the MAP subsystem. If infilename is omitted, control statements are assumed to be in-line directly following the MAP statement.
- mapfilename is the name of the file given to the MAP subsystem for output of the disk memory allocation tables. If mapfilename is omitted, no allocation tables are created. In this case the user can still process a layout description and obtain timing and layout analysis information as described below (see Section 5.5).

One complete MAP subsystem call produces one complete set of allocation tables for one I4DM layout. A layout is a collection of pages in a specified relationship to each other. When ALLOC assigns I4DM space for an area, it assigns one I4DM page for each logical disk page specified in the layout, and preserves the relationships among pages.

It is possible for a user to create two or more layouts with two or more MAP calls, use two or more ALLOC calls to assign I4DM space for them, and then use the assigned layouts concurrently. However, if this is done, the relationships between pages in different layouts will be unpredictable, although the relationships within each layout will be preserved.

Each layout is made up of areas (see below), and the areas are described by FORMAT statements in the MAP subsystem call. The following section discusses the FORMAT statement in detail.

5.3.1 THE FORMAT STATEMENT

The FORMAT statement is used to specify the arrangement on the logical disk of one area. An area is a named collection of user-specified pages on the logical disk. The FORMAT statement associated with the area identifies the pages in the area, specifies the arrangement of these pages on the logical disk, and assigns a name (the areaname) to the collection of pages.

The FORMAT statement has the following format:

```
FORMAT areaname,(formatspec)b̸
```

where

- areaname is a user-supplied name to be assigned to the area described in this FORMAT statement (up to six alphanumeric characters, with a letter for the first character).
- formatspec is a parenthesized sequence of operators that identify logical disk pages which are to be assigned to the area. formatspec defines the arrangement on the logical disk of these pages. formatspec is discussed in detail in the following section.

5.3.1.1 The formatspec

The MAP subsystem maintains a "current page position pointer," used in the processing of FORMAT statements, that always points to some logical disk band and some page within the band, in accordance with the logical disk of Figure 5-2. For each FORMAT statement, this pointer is initially set to Page 0, Band 0. The operators that make up the formatspec move the pointer in various ways. The operators will be considered in detail, starting with the operator used to reserve pages in the area.

5.3.1.2 The ±nP Operator

This operator has the form ±nP, where n is a signed integer. The n in this operator may be omitted if it is 1, and the sign may be omitted if it is positive.

The effect of ±nP is to reserve a set of n consecutive pages, starting with the current pointer position as the first page of the set and proceeding forward or backward on the logical disk (i.e., with or against the direction of physical rotation), depending on the sign of the operator. The pointer is left pointing to the next page on the logical disk, following the last page of the reserved set.

At the beginning of each formatspec, the pointer position is Band 0, Page 0. Thus the simple formatspec

(128P)

would specify an area consisting of 128 contiguous pages starting at Band 0, Page 0 and extending to Band 0, Page 127. The pages are contiguous because they are all within the same band.

5.3.1.3 The ±nS Operator

The n in this operator may be omitted if it is 1 and the sign may be omitted if it is positive. The ±nS operator moves the current page position pointer n pages forward or backward, depending on the sign, and has no other effect. The formatspec

(32S,64P)

would first move the pointer from its initial position at Page 0, Band 0 to Page 32; the 64P operator would then reserve Pages 32-95. The resulting area would consist of 64 contiguous pages starting at Page 32, Band 0 of the logical disk. The pointer would be left at Page 96, Band 0.

Another use of the ±nS operator is seen in the following example:

(32P,2S,32P)

Here 32 contiguous pages are reserved as part of the area, then the pointer is moved two pages forward, and another 32 contiguous pages are reserved.

5.3.1.4 Action of ±nP and ±nS Operators at a Band Boundary

Before discussing other operators, one more characteristic of ±nP and ±nS operators will be described. When either of these operators moves the pointer past Page 299 of any given band, the "next" page (for the purposes of the operator) will be Page 0 of the "next" band. The formatspec operators are implemented in this fashion to facilitate the handling of large data arrays.

The "next" band is the next band in numerical sequence counting from Band 0 through Band 51, with Band 0 following Band 51. This sequence is a notational property of the formatspec operators; as stated before, the bands of the logical disk are equidistant in terms of access time, and thus the sequence does not imply a physical relationship.

When a negative operator moves the pointer past Page 0 of any band, the "next" page will be Page 299 of the "previous" band.

For example, if the current pointer position is Page 290 of Band 13 and a 50P operator is encountered, Pages 290-299 of Band 13 and Pages 0-39 of Band 14 will be reserved, for a total of 50 pages. If the pointer position is Page 40 of Band 13 and a -50S operator is encountered, the pointer will be moved to Page 289 of Band 12.

5.3.1.5 The ±nB, ±nL, and ±nR Operators

The n in these operators may be omitted if it is 1 and the sign may be omitted if it is positive. These operators, like the ±nS operator, have no effect except to move the pointer; they provide additional convenience and flexibility by moving it in different ways.

The ±nB operator moves the pointer n bands from its current position, maintaining the pointer at the same page number within the new band. For example, if the current position is Page 234, Band 16 and a -4B operator is encountered, the resulting pointer position will be Page 234, Band 12. If a ±nB operator moves the pointer "past" Band 51 in a positive direction, it moves from Band 51 to Band 0, then to Band 1, etc. in sequence. If the pointer is moved in a negative direction "past" Band 0, it moves from Band 0 to Band 51, then to Band 50, etc. in sequence.

The ±nL operator resets the pointer to Page 0, Band 0, and then moves it to the nth page of the logical disk counting pages consecutively from Page 0 (Band 0). There is a total of 15,600 pages on the disk, and for purposes of the ±nL operator they form a continuous sequence, where Page 0, Band 0 is the 0th page on the disk and Page 299, Band 51 is the 15,599th page on the disk. Page 15,599 is "followed" in this sequence by Page 0. The n in a ±nL operator is taken modulo 15,600. Note that the pointer position resulting from a ±nL operator is independent of the previous position.

For example, the formatspec

(500L,10P)

moves the current page position pointer to Page 500 (i.e., Page 200 of Band 1) and then reserves ten pages, Pages 200 to 209 of Band 1. This sequence is independent of what precedes the L operator in the formatspec. For example, in the formatspec

(64P,2S,64P,500L,10P)

the 500L,10P operator sequence again reserves Pages 200 to 209 of Band 1.

The ±nR operator moves the pointer to the nth page within the current band, taking n modulo 300 and counting from Page 0 of the current band. The nth page within the band is counted from Page 0 of the band, regardless of the pointer position within the band before the operator is encountered. For example, if the pointer is in Band 10 and Page 100 when a 5R operator is encountered, the resulting position will be Page 5, Band 10. For purposes of this operator, the pages within a band are considered to "wrap around" from Page 299 to Page 0; thus a -5R operator, in the above example, would move the pointer to Page 295 of Band 10, and a 310R operator would move the pointer to Page 10 of Band 10.

5.3.1.6 Combinations of Operators

As noted above, operators are separated by commas and are interpreted and processed in sequence. Two or more operators may be enclosed in parentheses prefixed by a signed integer ±n; the parenthesized sequence of operators is repeated n times. For example,

3(2S,50P)

is exactly equivalent to 2S,50P,2S,50P,2S,50P. If the integer is negative, the signs of all operators in the parenthesized sequence are changed, e.g.,

-2(2S,-50P)

is exactly equivalent to -2S,+50P,-2S,+50P. Parenthesized sequences may be nested as in

2(2S,3(20P,2S))

which is exactly equivalent to 2S,3(20P,2S),2S,3(20P,2S) or 2S,20P,2S,20P,2S,20P,2S,2S,20P,2S,20P,2S,20P,2S.

5.3.1.7 Laying Out Two or More Areas

As explained above, each FORMAT statement in a sequence of statements to the MAP subsystem names and describes one area. Many layouts will require more than one area. This is achieved in a straightforward manner by using two or more FORMAT statements. Note, however, that each area in a single MAP call is described in terms of a Page 0, Band 0 which is a common reference point for all of the FORMAT statements. For purposes of any one FORMAT statement, this "(0,0)" is the initial position of the current page position pointer.

Areas are assigned physical space on the I4DM by the ALLOC subsystem. If two areas described in a single MAP call have certain pages of the logical disk in common, they will have corresponding pages of the physical disk in common after they are assigned, and the user must be aware of the implications.

5.3.1.8 The Continuation Character in FORMAT Statements

In a FORMAT statement, the formatspec may be very lengthy. The semicolon may be used as a continuation character within the formatspec. The semicolon should immediately follow a comma and should be followed by a carriage return.

Example
```
FORMAT INDATA,(3P,2S,4P,4S,6P,2S,20(S,P),2S,P,10(S,P),;ƀ
S,15(S,P),2S,10P)ƀ
```

5.3.1.9 Summary of the FORMAT Statement

The FORMAT statement is used in a MAP subsystem call to name and describe the arrangement on the logical disk of one area. Within one MAP subsystem call, all FORMAT statements use a common reference point (Page 0, Band 0 of the logical disk) for the area descriptions they contain. The description of an area is called a formatspec and is composed of operators. One operator (±nP) is used to reserve n consecutive pages in the area; other operators move a logical current page position pointer that is used to locate on the logical disk the sets of pages laid out by ±nP operators. The areas described in a FORMAT statement will be assigned space on the I4DM by the ALLOC subsystem.

5.3.2 DESCRIBING AN I4DM LAYOUT WITH THE MAP SUBSYSTEM

The following is a procedure for laying out an area:

(1) In two dimensions, lay out a 300 vertical by 52 horizontal grid (or a portion thereof). Each square in the grid represents a logical disk page.
(2) Number the horizontal axis from 0 to 51 (left to right).
(3) Number the vertical axis from 0 to 299 (top to bottom).
(4) The upper left-hand square then represents Page 0, Band 0 of the logical disk.

(5) Lay out the required page sets to be reserved for a data area for the ILLIAC IV process, marking them in sequence P1,P2,...,Pn, where Pn refers to a set of pages required in one access.

See Figure 5-3.

When laying out the reserved page sets, two rules should be kept in mind:

<u>Rule 1</u> When laying out a page sequence that crosses over from one band to another, and which is to be accessed in a single request, a two-page gap (+2S) should be left at the band crossover to allow for the electronic band switching. Conversely, a set of pages required in a single access should not cross a band boundary.

<u>Rule 2</u> Between any two successive accesses, a two-page gap should be left to allow for electronic switching.

<u>Note</u>: Combining these two rules, a single two-page gap is all that is required, for example, when reading from one band and then writing to another band.

When using FORMAT operators, two conventions should be remembered:

(1) Notationally, Band 51 is adjacent to Band 0.
(2) Logically, Page 299 in any band is contiguous to Page 0 in any other band. Notationally, Page 299 in any Band n is adjacent to Page 0 in Band n+1.

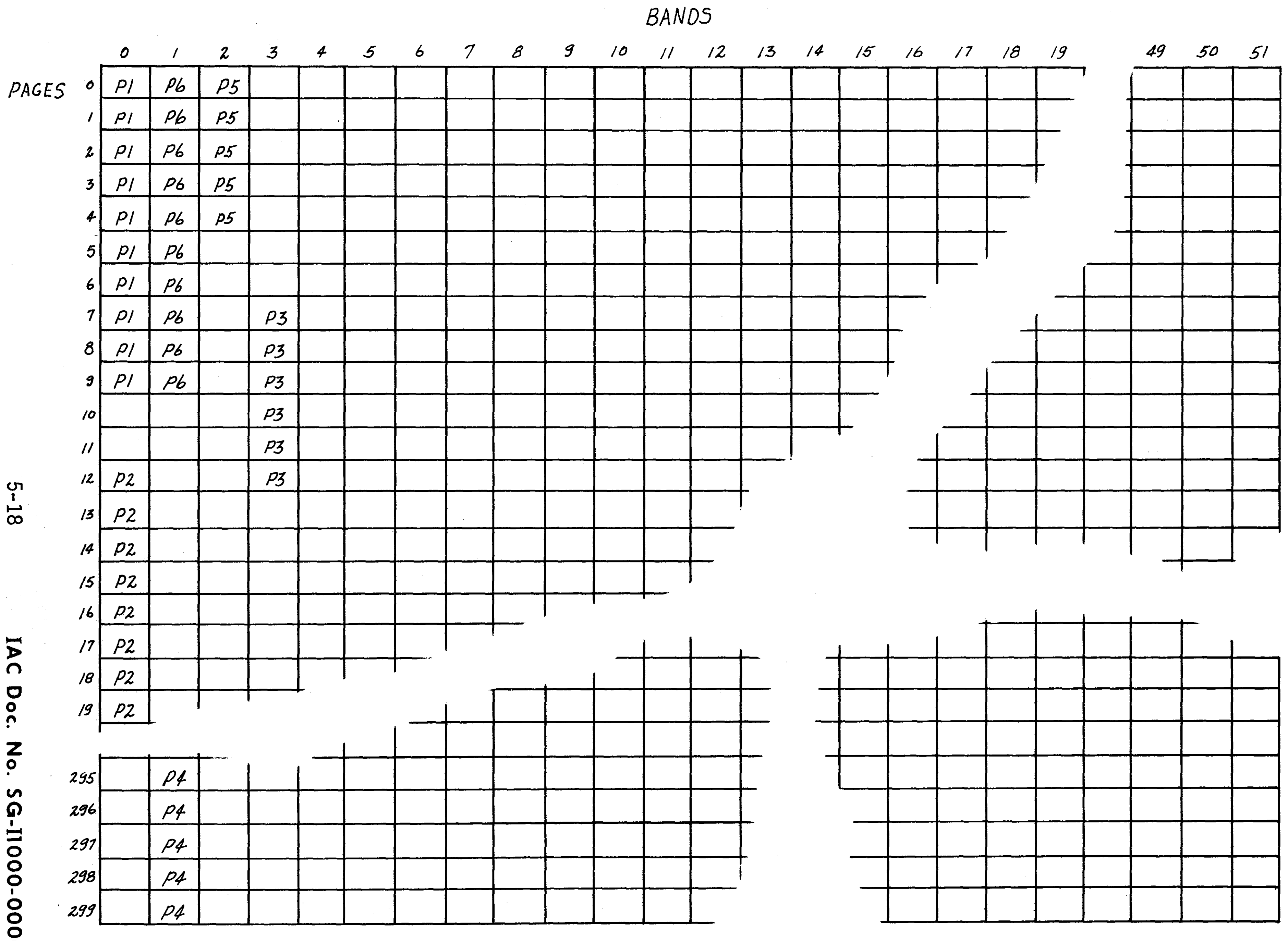


Figure 5-3. Working Disk Layout

5.3.3 AN EXAMPLE OF I4DM LAYOUT PROCEDURE

In this example, the FORMAT statement to specify the area illustrated in Figure 5-3 is constructed. The example is included to illustrate the use of the formatspec operators and is not meant to imply an efficient or typical layout. The areaname in this example will be EG1.

(1) The first ten-page set, denoted in the figure as {P1}, begins at Page 0, Band 0. The formatspec begins with the pointer implicitly at this position; therefore the operator

```
10P
```

will reserve the set.

(2) The next two pages are to be skipped, accomplished by the operator 2S. Therefore we have

```
10P,2S
```

(3) The set {P2} is reserved by the operator 8P:

```
10P,2S,8P
```

(4) The third set {P3} is located in Band 3. Therefore we have to move the pointer three bands with the 3B operator. The B operator maintains the pointer at the same relative page number (i.e., 19), so we have to move it back 12 pages to get the first page of the set {P3}:

```
10P,2S,8P,3B,-12S,6P
```

(5) The set {P4} also requires a band/page skip, so we use the L operator. The first page of {P4} is at Page 595 (counting pages from Page 0, Band 0). Therefore the operator 595L

positions the pointer at the first page of {P4}:

```
10P,2S,8P,3B,-12S,6P,595L,5P
```

(6) The set {P5} is notationally adjacent to {P4} using the P operator (since Page 299 of Band 1 is adjacent to Page 0 of Band 2). Therefore, the 5P operator used above to reserve {P4} can be changed to 10P to reserve both {P4} and {P5}:

```
10P,2S,8P,3B,-12S,6P,595L,10P
```

(7) The set {P6} is back one band from {P5}. Therefore the -B operator is required. Remembering that the relative page position remains the same with the B operator, we can move to Page 0 with the R operator:

```
10P,2S,8P,3B,-12S,6P,595L,10P,-B,0R,10P
```

This string when enclosed in parentheses is a complete formatspec and can be used in a FORMAT statement:

```
FORMAT EG1,(10P,2S,8P,3B,-12S,6P,595L,10P,-B,0R,10P)b̸
```

5.4 RELATIONSHIP BETWEEN A LOGICAL DISK LAYOUT AND AN I4DM SPACE ASSIGNMENT

When a MAP subsystem call has been completed and a file of allocation tables produced, the ALLOC subsystem is used to allocate I4DM space for the layout of the one or more areas described in the allocation tables. In doing this, ALLOC assigns one I4DM page for each logical disk page reserved in the MAP layout.

There is thus an exact one-to-one correspondence between the reserved logical disk pages and the assigned I4DM pages. The essential relationships among reserved logical disk pages are preserved in the arrangement of assigned I4DM pages; these relationships are the contiguity of pages within page sets and the rotational delays between page sets.

The following section describes the resulting mapping of the user's assigned I4DM space.

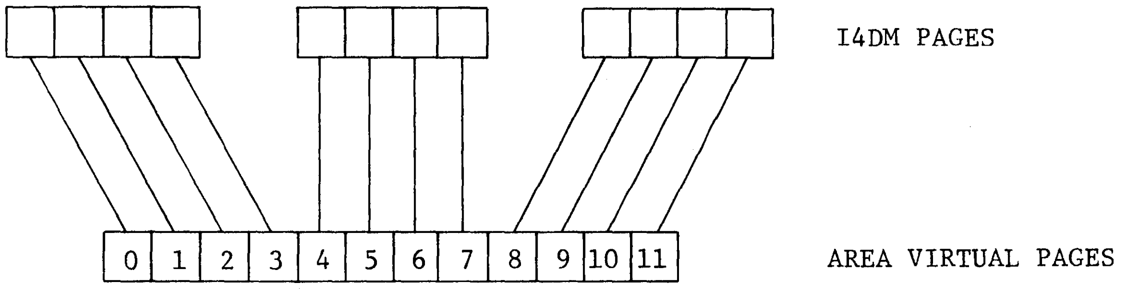
5.4.1 THE AREA AS A VIRTUAL I4DM MAP

A FORMAT statement specifies not only an areaname and a collection of pages, as described previously, but also a sequence of pages. The area page sequence is the sequence in which the pages are reserved in the formatspec: Page sets are sequenced in the exact order in which they are specified in the formatspec by ±nP operators, and within each contiguous page set the individual pages are ordered either positively (i.e., with the rotation of the disk) or negatively (against the rotation of the disk) according to the sign of the nP operator.

Observe that this sequence is directly determined by the formatspec used to describe the area, and is not dependent on the sequence in which the pages occur on the logical disk, since the formatspec operators permit sets of pages to be reserved anywhere on the logical disk in any order.

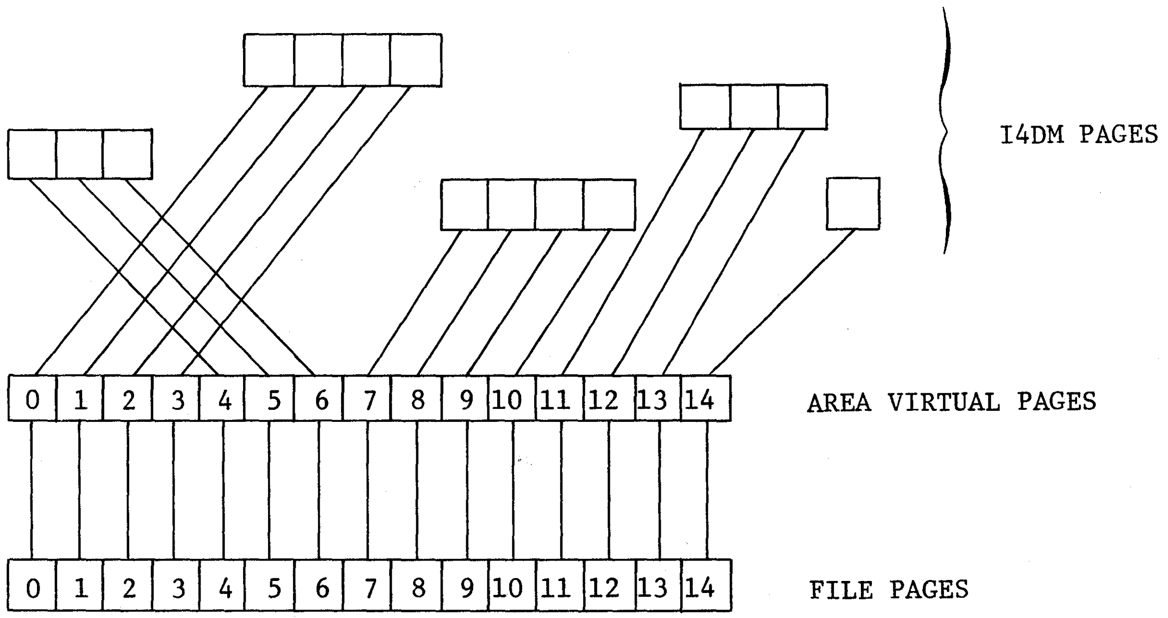
The pages collected in an area are "virtually" arranged in a consecutive order according to this sequence, with the areaname pointing to the first page in the sequence (Page 0 of the area) and each subsequent page addressed relative to Page 0 of the area. In this sense, then, an

I4DM area is a virtual I4DM map. The following diagram illustrates (for a simple case) the correspondence between a collection of disk pages and the virtual page sequence of an area.



5.4.1.1 The Area Virtual Map and the COPY Subsystem

When data is transferred from an I4DM area to a file by means of COPY, the COPY subsystem reads I4DM pages in the sequence specified by the area virtual map and writes file pages in sequence starting with Page 0 of the file. In the inverse operation, where data is transferred from a file to an I4DM area, COPY reads file pages in strict sequence and writes I4DM pages in the sequence specified by the area virtual map, as shown in the following diagram for a case where the file and the I4DM area contain the same number of pages.

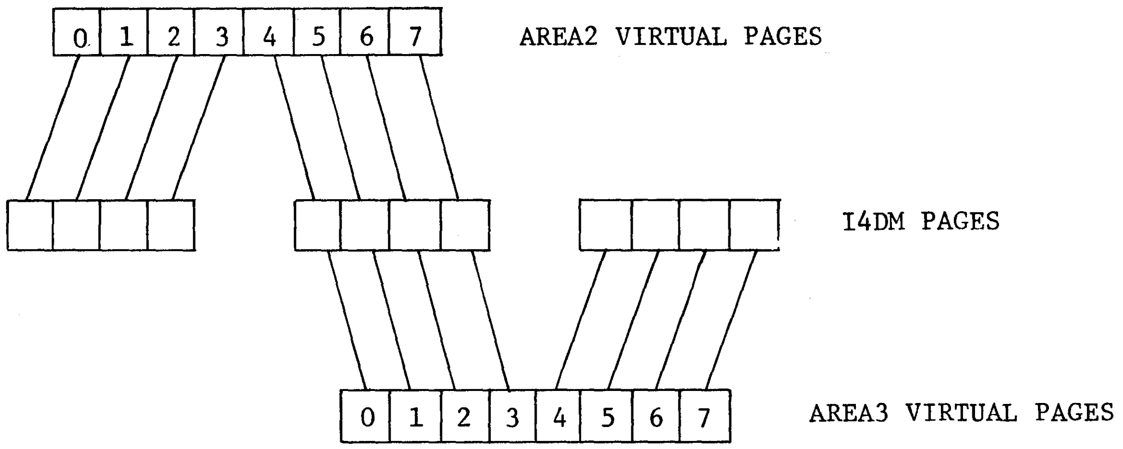


5.4.1.2 Examples of Virtual Maps

An I4DM page may correspond to more than one virtual page, but every virtual page corresponds to exactly one I4DM page. For example, consider the areas described in the following FORMAT statements (assume that these FORMAT statements were processed in the same MAP call):

```
FORMAT AREA2,(4P,2S,4P)ƀ
FORMAT AREA3,(6S,4P,2S,4P)ƀ
```

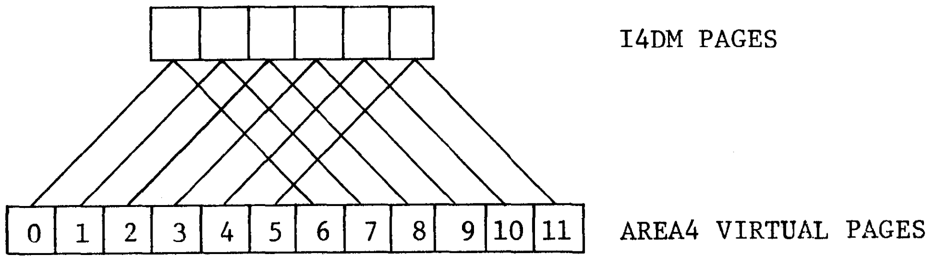
The first of these statements reserves Pages 0-3 and 6-9 (Band 0) of the logical disk; the second reserves logical disk Pages 6-9 and 12-15 (Band 0). Thus logical disk Pages 6-9 (Band 0) are reserved <u>twice</u>, and this fact is preserved when I4DM pages are assigned to the two areas by ALLOC. The following diagram shows the resulting correspondences.



Finally, note that a logical disk page may correspond to more than one virtual page in the same area. Consider the FORMAT statement

FORMAT AREA4,(6P,-6S,6P)␢

The effect of the statement is to reserve logical disk Pages 0-5 (Band 0) twice within the same area. ALLOC will assign six contiguous I4DM pages, each corresponding to two different virtual page numbers in AREA4, as shown in the following diagram.



A six-page file could then be transferred to AREA4 via COPY, with the result that each of the six pages of information could be addressed in I4DM by two different virtual addresses.

5.5 THE MAP SUBSYSTEM LIST OUTPUTS

There are two operators that can be used in a FORMAT statement which have not yet been described. These are:

X — which means mark the current page for printing;
T — which means mark the current page for timing.

"Current" in the above definitions refers to the current position of the page pointer.

Note that unlike the other operators, X and T are not prefixed with a signed integer. The effect of X or T is to mark the current page for printing or timing, without moving the pointer. Thus the formatspec

(X,20P,2S,X,30P)

reserves two sets of 20 and 30 pages respectively, with a two-page gap between them, and the first page of each set is marked for printing. To mark all reserved pages for printing, in the same layout, one would write

(20(X,P),2S,30(X,P))

The construction X,T or T,X marks the current page for both timing and printing. The following formatspec again reserves two sets of 20 and 30 pages respectively, with a two-page gap between them, and marks the first page of each set for timing and all reserved pages for printing:

(T,20(X,P),2S,T,30(X,P))

Note: It is syntactically permissible to prefix an X or T operator with a signed integer, e.g., 5T. However, the signed integer in this case is ignored, and 5T is exactly equivalent to T.

5.5.1 THE PRINT STATEMENT

The PRINT statement has the following format:

```
PRINT areaname1,areaname2,...,areanamenƀ
```

where

- areaname(s) have been previously specified with FORMAT statements in the MAP call.

The PRINT statement causes MAP to produce a diagram representing the logical disk, and indicating all pages marked for printing in preceding FORMAT statements. The diagram consists of 52 columns and 100 rows; each column represents a band (with Band 0 at the left), and each row entry within a column represents three contiguous pages within the band (with Pages 0, 1, 2 at the top of each column). The three pages are internally represented by a pattern of three bits – 100 if the first of the three pages is marked for printing, 110 if the first and second are marked, etc. On the PRINT diagram, each entry consists of the corresponding octal number, i.e., 0 if none of the three pages is marked, 3 if the second and third are marked, etc.

5.5.2 THE TIME STATEMENT

The TIME statement has the following format:

```
TIME areaname1,areaname2,...,areanamenb̸
```

where

- areaname(s) have been previously specified with FORMAT statements in this MAP call.

The TIME statement writes out the I4DM rotational time delay between pages marked for timing in preceding FORMAT statements. Time delays will be computed between pages in the sequence in which they occur on the logical disk, and not in the sequence of the virtual map.

5.5.3 THE END STATEMENT

The END statement has the following format:

```
ENDb̸
```

An END statement is required as the last control statement to the MAP subsystem. Upon recognizing the END statement, MAP will process all the other control statements in the sequence and will produce a file of allocation tables, TIME output, and/or PRINT output, as previously specified by the user.

5.6 THE ALLOC SUBSYSTEM

The MAP subsystem does not assign any actual I4DM space, but merely produces a file of allocation tables for the areas specified in FORMAT statements. The ALLOC subsystem, using these tables, assigns disk space in I4DM to an ILLIAC IV user.

The ILLIAC IV is a resource shared by a community of users. Although users may not concurrently use the ILLIAC IV Processor, the I4DM component of ILLIAC IV may be shared. Thus physical I4DM space may be assigned to more than one user. The ALLOC subsystem examines the current status of the I4DM and assigns all of the requesting user's areas, as described to MAP, if space is available on the I4DM. This assignment is all or nothing for each ALLOC call.

The ALLOC statement has the following format:

```
ALLOC mapfilename{,allocid}ƀ
```

where

- mapfilename is the name of an output file from the MAP subsystem, containing allocation tables for all of the areas described in one call to the MAP subsystem.
- allocid is an optional identifier of this call to the ALLOC subsystem. This allocid must be used in a subsequent call to the DALLOC subsystem, if such a call is made; if the user does not intend to make a subsequent call to DALLOC, the allocid may be omitted from the ALLOC statement.

The ALLOC subsystem will attempt to assign all of the space called for by the allocation tables in the mapfilename. If this is impossible, it will assign no space at all — it will not assign partial space, nor will it alter the specified layout to fit it into the available space.

Assuming the ALLOC succeeds in allocating all the necessary space, this space is then identified with the jobid given to the job in which the ALLOC call occurs (see SUBMIT, Sections 4 and 8), and with the allocid, if an allocid was included in the ALLOC statement.

The space remains assigned until marked for release on termination of the job or when the DALLOC subsystem (see below) is called. Observe that the ALLOC call must be included in the same job as the user's RUN statement (see RUN, Sections 7 and 4) since assigned space is marked for release on job termination.

5.7 THE DALLOC SUBSYSTEM

The user may release space that has been previously allocated in the same job by using the DALLOC statement, with the following format:

```
DALLOC allocidƀ
```

where

- allocid is the same allocid supplied by the user in the ALLOC statement that was used to allocate the space that is now to be released.

SECTION 6

COMPLETE ACL STATEMENT FORMAT

SECTION 6

COMPLETE ACL STATEMENT FORMAT

CONTENTS

SECTION 6

COMPLETE ACL STATEMENT FORMAT

6.1 INTRODUCTION

The complete ACL statement format is as follows:

```
{label:}SUBSYSTEM-NAME argument1,argument2,...,argumentnɋ
```

where

- label is a name that can be used as an argument of a TEST statement; any labelled ACL statement can be conditionally branched to by a TEST statement (see Section 7). The label, if used, must be followed by a colon (:). The colon may optionally be followed by one or more blanks (spaces or tabs).
- SUBSYSTEM-NAME is the name of the ACL subsystem called by the statement.
- argument is information supplied by the user. The rules for specifying arguments are discussed in Section 3. Classes of arguments are discussed in Section 6.2 below.
- ɋ is a carriage return, and terminates the statement (but see Section 6.4 below).

6.2 CLASSES OF ARGUMENTS

Most of the arguments specified in an ACL statement can be grouped into one of several classes. Since each class has its own standard syntax, the rules for the construction of arguments are discussed separately for each class. The following classes of arguments are described in this section:

- Filenames
- Values
- Macro-calls
- Control Parameters
- Expressions

Since some arguments do not fall into any of the above classes, the rules of construction for these arguments are presented in the description of the particular ACL statement where they are used (Section 7).

6.2.1 FILENAMES

A <u>filename</u> has the following format:

```
<directoryname>name{.extension}{;version}
```

6.2.2 VALUES

A <u>value</u> is any signed decimal or octal integer, and is represented in the computer as a signed 36-bit word. An octal integer is prefixed by a # character; thus "#27" means "27 octal," while "27" means "27 decimal."

A <u>value</u> may be used as an argument in the EQU statement (see Section 7), as a formal argument in a macro-call (see Section 8), and as an element in an <u>expression</u> (see Section 6.2.5).

6.2.3 MACRO-CALLS AS ARGUMENTS

It is a general rule that any argument in an ACL statement may be replaced by a <u>macro-call</u>. This is not explicitly noted in the syntax descriptions for individual statements, but it is always the case.

A <u>macro-call</u> is defined as consisting of the name of a previously defined macro optionally followed by a parenthesized list of actual arguments — i.e., the complete calling sequence of the macro:

```
macroname(arg1,arg2,...,argn)
```

For further details, see Sections 7 and 8.

6.2.4 CONTROL PARAMETERS

A control parameter is a name consisting of up to six alphanumeric characters with an assigned value. A control parameter can be defined and a value assigned to it by only one means, namely, the EQU statement (see Section 7 for details).

A control parameter can be used as an argument in the TEST or EQU statement or as an actual argument in a macro-call. Additionally, it may be used as an element in an expression (see Section 6.2.5 below).

6.2.4.1 The STATE Parameter

The STATE parameter is a special control parameter that is predefined by the system and normally has a binary value assigned to it by the user's most recent subsystem call. Various bits are used to signal various conditions. At present the only consistent meaning of STATE is that STATE will be negative if the subsystem called by the user's most recent ACL statement aborted "involuntarily" – i.e., if the subsystem was interrupted and aborted by a higher fork of the system.

Two ACL subsystems, EQU and TEST, do not assign a value to STATE but leave its previous value unchanged.

The user may assign a new value to STATE at any time, by using an EQU statement; however, the system will assign it another value as soon as it processes the next ACL statement (except EQU or TEST).

6.2.5 EXPRESSIONS

An expression consists of one or more elements in combination with operators. An element may be either a value, a control parameter, or an expression enclosed in square brackets ([]). Elements must be separated by operators. The operators are defined by the following table.

Type	Precedence	Symbol	Function
Unary	1	+	Gives the element following a positive value.
	1	-	Gives the element following a negative value.
	1	*	Complements the element that follows.
	1	#	Prefix for an octal number.
Shift	2	←n	Shifts the preceding element n bits to the left (end-off shift with zero fill on 36-bit word).
	2	\n	Shifts the preceding element n bits to the right (end-off shift with zero fill on 36-bit word).
Logical	3	&	Bitwise logical AND.
	3	@	Bitwise logical inclusive OR.
	3	$	Bitwise logical exclusive OR (XOR).
Arithmetic	4	*	Multiply.
	4	/	Divide (returns integer part of quotient).
	5	+	Add.
	5	-	Subtract.

The following are examples of valid expressions.

Example 1 A*[B+C]

Here A, B, and C are previously defined control parameters. The sum of B and C is multiplied by A and the result (a signed integer) is the value of the expression. Note that square brackets are used in the same fashion as parentheses in ordinary algebraic notation.

Example 2   [STATE←4]\35

The value of the system-defined control parameter, STATE, is first shifted left four bits and then shifted right 35 bits. Values are represented in the computer as 36-bit words; therefore, after the specified shifts have taken place only the original Bit 4 of STATE (fifth bit from the left) remains, and occupies the rightmost position in the word. The rest of the word is filled with zeroes. Thus the value of the expression is 1 if Bit 4 of STATE is 1 and 0 if Bit 4 of STATE is 0. Note: The value assigned to STATE is unaffected. An expression cannot assign a new value to a control parameter; only an EQU statement can do this.

Example 3   #020000000000&STATE

This is similar in effect to Example 2. The octal number is used as a mask, i.e., it is ANDed with STATE to pick out Bit 4 of STATE. The value of the expression will be 0 if Bit 4 of STATE is zero, and positive if Bit 4 of STATE is a one. (Specifically, it will be #020000000000 in the latter case.)

When combining elements in an expression, the system performs the operations in the order of precedence shown in the table. The unary operations are performed first; the shifts are done next, followed by the logical operations. Arithmetic operations are then done from left to right, with multiplications and divisions performed first. (In division, fractional parts are truncated.) Additions and subtractions are then performed, left to right.

Expressions are only evaluated by the EQU and TEST statements; in the EQU statement an expression may be used as the second argument and its value assigned to the control parameter used as the first argument, and in the TEST statement its value is used as the condition for a branch (see Section 7 for details). An expression may also be used as the text of a macro that is intended to be referenced (see Section 7).

6.3 USE OF A MACRO-CALL IN PLACE OF AN ACL STATEMENT

If a macro has been defined and its text consists of one or more legal ACL statements, a macro-call to that macro can be entered in the stream of ACL statements as if it were itself an ACL statement.

6.4 CONTINUATION AND COMMENT CONVENTIONS

In Section 3.5 two rules are given for the use of the semicolon as a continuation or comment character:

(1) When placed immediately after a comma that terminates an argument, or immediately following the blank(s) after the subsystem name, the semicolon is a continuation character; it may be followed by a carriage return and the statement.

(2) When used as the first character on a line that is not a continuation of the previous line, the semicolon is a comment character; it causes the entire line up to the carriage return to be interpreted as a comment. The comment may be any string of characters, terminated by the carriage return.

Rule (2) is correct and complete as it stands, but Rule (1) may be augmented by two additional rules to show that a comment can be entered on the same line as a statement or part of a continuing statement.

(3) When a statement is continued as in Rule (1), the carriage return need not follow the semicolon immediately; a comment may be inserted between the semicolon and the carriage return.

Example

```
DEL PROG.ONE,PROG.TWO,PROG.THREE,;THIS IS A COMMENTb̸
PROG.FOUR,PROG.FIVEb̸
```

(4) After the last argument of a statement, a comma may be added followed by a semicolon. The semicolon functions as in Rule (3), and a comment can be inserted between the semicolon

and the carriage return. However, the semicolon also is a continuation character under Rule (1), and consequently an additional carriage return is required to terminate the statement.

Example

```
DEL DATA.ONE,; I DONT NEED THIS FILE ANYMOREƀ
ƀ
RENAME DATA.NEW,DATA.ONEƀ
```

6.5 LEADING AND TRAILING BLANKS IN ARGUMENT FIELDS

Any blanks (spaces or tab characters) preceding or following an argument within the commas that delimit the argument (or between the argument and a carriage return) are ignored. This applies to single blanks and also to strings of any number of blanks.

A semicolon used as a continuation or comment character is technically an argument under the general syntax rules of ACL. This means that a semicolon used in this fashion may be immediately preceded or followed by a string of blanks and will still work.

6.6 QUOTATION MARKS

Quotation marks enclosing an argument prevent the argument from being scanned for break characters, which include blanks, carriage returns, and semicolons. Thus an argument in quotation marks can include break characters.

Examples

```
";argone"
"exp 4"
```

6.7 CONTROL CHARACTERS

Control characters are indicated in this text by a prefixed up-arrow: for example, ↑A means "control A". On terminals that have a CONTROL key, a control character is input by simultaneously depressing the CONTROL key and some other key — e.g., ↑A is input by simultaneously depressing the CONTROL key and the A key.

Five control characters are usable in ACL. The characters ↑A, ↑Q, and ↑R are used for editing during input of ACL statements; ↑T and ↑C have different functions which are explained separately below. As a rule, control characters are only useful in interactive mode.

6.7.1 EDITING STATEMENTS DURING INPUT

↑A, entered within a statement, causes the last previous character in the statement to be deleted from the input. Thus if the user accidentally strikes the wrong character while entering a statement, he may follow it with a ↑A, then type the correct character and proceed with the rest of the statement.

↑Q causes the entire line currently being typed to be deleted from the input. The user may then start over from the beginning of the line.

↑R causes the entire current line to be retyped. This is useful if the user has repeatedly used ↑A in the line, and it has become difficult to read. After the line has been retyped, the user may begin typing again where he left off.

6.7.2 THE ↑T CHARACTER

↑T, entered between ACL statements, causes the system to type out status information on the ACL subsystem currently being executed (if any).

6.7.3 THE ↑C CHARACTER

Normal exit from ACL subsystems is by means of an END control statement (see Section 3.5). In abnormal situations, an executing subsystem can be halted in a disorderly fashion by entering a ↑C. The results of this action are unpredictable and the user is cautioned to use ↑C only as a last resort in abnormal situations when END cannot be used (e.g., if a subsystem is obviously caught in a loop).

SECTION 7

COMPLETE SET OF ACL STATEMENTS

SECTION 7

COMPLETE SET OF ACL STATEMENTS

CONTENTS

SECTION 7

COMPLETE SET OF ACL STATEMENTS

The following is a summary of the formats of all ACL statements. Sections containing full discussions of statements follow the summary in alphabetical order.

Note that pages are not numbered consecutively in Section 7; instead each set of pages dealing with one ACL statement is numbered independently, as is the following summary.

SUMMARY OF ACL STATEMENTS

`ALLOC mapfilename{,allocid}ƀ` — Assign ILLIAC IV disk memory space to the user as specified in the file named mapfilename.

`ALT filenameƀ` — Use the file named filename as an alternate input source of ACL statements.

`ASK infilename,{outfilename},{listfilename}{,MACRO=macrofilename}ƀ` — Assemble source code from file named infilename, put object code in file named outfilename, and listing in file named listfilename.

`CKPOINTƀ` — Description to be supplied.

`COPY sourcefile,destinationfileƀ` — Copy contents of file designated by sourcefile to file designated by destinationfile. Both files must be local; sourcefile may be in another user's directory.

`CPYNET sourcefile,destinationfile{,BYTE=bytesize}{,TYPE=typespec}ƀ` — Data transfer across the ARPA Network, between the local system and a remote Host.

`DALLOC allocidƀ` — Mark for deallocation all ILLIAC IV disk memory space previously allocated within same job and associated with same allocid.

`DEDƀ` — Call the text editor subsystem, DED.

`DEL filename1,filename2,...,filenamenƀ` — Delete the specified file(s) from the system and the filename(s) from the user's file directory.

`DELJOB jobid,userid,passwordƀ` — Abort the specified job (if it is being processed) and delete it from the batch queue.

DIR {filename1,filename2,...,filenamen}b̸ — List the user's file directory, or the entries for specified files.

EQU name,expressionb̸ — Define the control parameter name and assign the value expression to it.

GLYP infilename,outfilename,{listfilename}{,ASSEMBLE=yesno}b̸ — Compile GLYPNIR source code from file named infilename, put relocatable ASK object code in file named outfilename, and listing in file named listfilename.

HELPb̸ — Answer questions typed in by the user.

INQ {jobid}b̸ — Returns status information on specified job or on all jobs under a given userid.

LIB libname,filename1,filename2,...,filenamenb̸ — Construct a user library of files of relocatable ASK code for use by the link editor.

LINKED {infilename},outfilename{,OPTIONS=optionstring}b̸ — Convert relocatable ASK object-code files to link-edited load modules and put load modules in file named outfilename. Take control statements from file named infilename (object-code files are specified by control statements).

LOGIN userid password accountb̸ — Enter user into system (not a genuine ACL statement). Returns ACL prompt.

LOGOUTb̸ — In a batch job, means terminate batch job. In an interactive session, means terminate session. Does not disconnect user from ILLIAC IV System.

MACRO name{,(arg1,arg2,...,argn)}b̸ — Define following text as an ACL macro with name name and formal arguments arg1, arg2, etc.

`MAP {infilename}{,mapfilename}ƀ` — Define a layout for data areas in ILLIAC IV Disk Memory, using control statements from file named infilename. Construct allocation tables in file named mapfilename.

`MAXAFSƀ` — Type out the number of pages of disk space assigned to the user for file storage.

`MOVE source,destinationƀ` — Transfer data between a file in the user's directory and an I4DM area.

`NEWSƀ` — Type out news about ACL subsystems.

`NOTIFY "message"ƀ` — Route message to system operating staff.

`OUTPUT {LINE=mode}{,WIDTH=linewidth}ƀ` — Select full or half duplex terminal operation and set width of output line on user terminal.

`RENAME oldfilename,newfilenameƀ` — Change oldfilename to newfilename in user's file directory.

`RUN infilename,{DMPFIL=dumpfilename}{,MAXTIM=time}ƀ` — Transfer load modules from file named infilename to ILLIAC IV Disk Memory, load root segment into PE memory, and begin execution.

`SENDƀ` — Send a message to a specified set of ACL users.

`SSK infilename,listfilename,{VALUE=taskvalue},{arealist}{,MAXTIM=time}ƀ` — Simulate the object code contained in file named infilename with the ILLIAC IV Simulator, SSK.

`SUBMIT pifname,{pofname},{runcode},userid,password,accountƀ` — Enter a request into the batch queue for processing of a file of ACL statements named pifname.

TEST expression,({neglabel},{zerolabel},{poslabel})b̸ — Branch conditionally on whether value of expression is negative, zero, or positive.

UDIRb̸ — List a directory of active files in the user's directory that have been stored on the UNICON, or the entry for a specified active file that has been stored on the UNICON.

The ALLOC subsystem assigns space on the ILLIAC IV Disk Memory according to the user layout specifications.

```
ALLOC mapfilename{,allocid}ƀ
```

where

- mapfilename is the name of the output file from the MAP subsystem which contains a description of the user's required layout (see MAP).
- allocid is a user-supplied identification that must be supplied if a DALLOC statement is to be used later in the same job (see DALLOC). allocid can be any six-character numeric field.

Description: The ALLOC subsystem absolutely assigns ILLIAC IV Disk Memory pages according to the relative layout specified in file mapfilename. The assignment of space by ALLOC is "all or nothing"; that is, either the entire set of pages requested by the user is assigned or no assignment is made. The space assignments made by ALLOC are linked to both the jobid and the mapfilename so that they may be deallocated (marked for reassignment) by either of the following:

(1) a termination of job,
(2) a DALLOC subsystem call (see DALLOC).

Disk memory space assignments are marked for reassignment whenever the associated job terminates, either normally (by an EOF or LOGOUT) or abnormally. The user should always call ALLOC, COPY, and RUN in sequence, in the same job.

The ALT statement changes the ACL input source to a specified file of ACL statements, and causes a return to the main sequence on completion of processing of the file.

```
ALT filenameђ
```

where

- filename is the name of a file in the user's ILLIAC IV directory containing a sequence of ACL statements.

Description: ALT causes the input ACL statement stream to be taken from the file specified by filename rather than from the current source (e.g., the user's terminal or an ACL batch file). Any predefined sequence of legal ACL statements stored in a file may be initiated by use of the ALT statement specifying the filename. On completion of processing of the file of ACL branched-to with the ALT statement, processing continues with the ACL statement immediately following the ALT statement in the original sequence.

Any ACL statement file may include an ALT statement to any other ACL statement file.

The control parameters referenced in an ACL statement file are globally defined; that is, they may be defined within the file, or within any previously processed ACL statement within the same job sequence.

The ALT statement may be used interactively or in a batch ACL sequence.

Example:

```
ALT ALPHA.JQS↵
```

ACL statements are to be taken from a file named ALPHA.JQS.

The ASK statement is a request for assembly of an ASK source-code program. It can only be used in batch mode.

```
ASK infilename,{outfilename},{listfilename}{,MACRO=macrofilename}ƀ
```

where

- infilename is the name of the file containing ASK source code to be assembled.
- outfilename is the name of the file for the assembled, relocatable object code. If outfilename is omitted, the default filename for the output file is the same as the infilename with REL as the extension. If an outfilename is entered without an extension, the default extension REL is used.
- listfilename is the name of the file for the ASK listing of the source program. If listfilename is omitted, no listing will be produced.
- macrofilename is the name of a user file of ASK macros that are called in the ASK source code in the file named infilename.

Description: The ASK statement is a request for a B6700 assembly of ASK source code stored in the file named infilename. Since it requires the B6700 resource, the ASK statement can only be used in a batch file. Syntax and other error messages generated by ASK will be placed in the user's primary output file (see SUBMIT). If the assembly is successful, the relocatable object code will be written in the file named outfilename in the user's directory. A listing of the source program with whatever assembler directives have been included in the file named infilename will be transferred back to the user's directory in a file named listfilename.

The COPY statement causes file transfers within the local system. In particular, it may be used to copy a file from one user's directory to another.

```
COPY sourcefile,destinationfile↵
```

where

- sourcefile designates the file to be copied. This may be an active file in the requesting user's directory, in which case it has the usual filename format:

  ```
  name.extension;version
  ```

  Alternatively, it may be an active file in another user's directory, with the format

  ```
  <directoryname>name.extension;version
  ```

  where directoryname is the identifier (commonly a userid) associated with the file directory in which the file is held. In either case, version may be omitted, in which case the highest numbered version of the file will be used.
- destinationfile designates the file to be copied to. This must be a file in the requesting user's directory, with the usual format:

  ```
  name.extension;version
  ```

  The version may be omitted, in which case a new version will be created if there is already a file or files with the same name and extension.

Description:

COPY transfers the contents of sourcefile to destinationfile. The name and contents of sourcefile remain unaltered. If COPY is successful, a message is returned to the user indicating the number of bytes transferred. If unsuccessful, COPY returns an appropriate diagnostic message.

Note:

Certain files have "protected" status, and cannot be accessed by a user who is not logged in under the proper userid. If such a file is designated as sourcefile, COPY will fail.

Examples:

```
COPY <JONES>ALGORITHM.SOURCE,PROGRAM1.SOURCEƀ
```

Transfer the contents of file ALGORITHM.SOURCE in user Jones' directory to file PROGRAM1.SOURCE in the user's directory.

```
COPY PROGRAMQ.ASK,PROGRAMX.ASKƀ
```

Transfer the contents of file PROGRAMQ.ASK in the user's directory to file PROGRAMX.ASK, also in the user's directory. The user will then have two files with different names but identical contents.

The CPYNET statement causes file transfers between the local system and a remote Host.

```
CPYNET sourcefile,destinationfile{,BYTE=bytesize}{,TYPE=typespec}b̸
```

where

- sourcefile is the designation of the file whose contents are to be copied.
- destinationfile is the designation of the file resulting from the COPY action.
- bytesize specifies the size of byte to be used and may be specified as 8, 32, or 36. In general, there is no need to specify the byte size, and if BYTE=bytesize is omitted, CPYNET will use an appropriate byte size depending on the particular remote Host involved.
- typespec specifies the type of file transfer to be performed, as follows:

TYPE=A	transfer an ASCII file. This is used for text files and is the default option if TYPE=typespec is omitted.
TYPE=I	transfer an image file. This is used for binary files.
TYPE=L	used where the remote Host has a word size not equal to 36 bits.

Legal Argument Combinations:	One of the two required arguments (sourcefile or destinationfile) must designate an active file in the local system, and the other must designate a file at a remote Host.
	The designation of the local file is in the following format:

```
name{.extension}{;version}
```

i.e., a filename subject to the rules given in Section 3. If version is omitted and sourcefile is designated, then the highest (newest) version of the file is selected. If version is omitted and destinationfile is designated, then a new version is created.

The designation of the remote file is in the following format:

```
(filename,hostid,userid{,password{,account}})
```

where

filename is the name of the file at the remote Host.

hostid is either the name of the remote Host or the corresponding octal number.

userid is the user identification under which the file is held at the remote Host.

password is the password (if any) associated with the userid at the remote Host, and should be omitted if not required.

account is an accounting number to be used at the remote Host, and should be omitted if not required. If password is omitted, account must also be omitted.

The parentheses in the above format are mandatory.

Description:

CPYNET makes use of file-transfer programs that conform to the ARPA Network File Transfer Protocol (FTP), and the details of its action (byte handling etc.) are largely dependent upon the FTP conventions for the particular remote Host being accessed. These details are published by the various Network Hosts.

CPYNET gives the appropriate commands to the file-transfer programs to cause the contents of the sourcefile to be transferred to the destination-file. If the transfer is successful, CPYNET returns a message to the user indicating the number of bytes transferred; if it is unsuccessful, it returns an appropriate error message.

Examples:

```
CPYNET (MDATA,MOON,JONES,XYZ,123),DATA.INPUTb̸
```

Transfer the contents of file MDATA, held under userid JONES at the Moon Host, to file DATA.INPUT in the user's directory in the local system. Jones' password at the Moon Host is XYZ, and the account number used is 123.

```
CPYNET DATA.TRANS,(TDATA,MOON,JONES,XYZ,123)b̸
```

Transfer the contents of file DATA.TRANS in the user's directory in the local system to file TDATA under userid JONES at the Moon Host.

The DALLOC subsystem releases (marks for deallocation) the set of ILLIAC IV Disk Memory pages assigned to the user's job.

```
DALLOC allocidƀ
```

where

- allocid is the same name used as allocid in a preceding ALLOC statement in the same job.

Description: The DALLOC subsystem should be used when a user wishes to release an ILLIAC IV Disk Memory space assignment within a job.

The ALLOC subsystem will automatically deallocate disk memory space assigned to the user's job when the job terminates. Therefore, DALLOC is only required when a user wishes to release a space assignment within a job.

The DED statement calls the text editor subsystem, DED (see Appendix A).

```
DEDƀ
```

Description: DED is a simple text editor, designed to be used in the interactive mode. The input to DED may be either characters typed on the user's terminal or a file in his directory. The contents of a file may be referenced by line numbers or by a character-string search. The output is made a file in the user's directory. For a full explanation of DED, see Appendix A.

The DEL statement deletes the specified file(s) from the user's active file space and the specified filename(s) from the user's file directory.

```
DEL filename1,filename2,...,filenamen↵
```

where

- each <u>filename</u> is the name of an active file in the user's file directory.

Caution: The user must not delete files that are specified in a batch ACL file which the user has submitted (see SUBMIT) but which has not yet completed processing.

The DELJOB statement is used to delete a specified job from the batch queue. If the job is being actively processed, the DELJOB statement will have no effect. To abort a job during processing, the user must send a message to the system operator via a NOTIFY statement.

```
DELJOB jobid,userid,password␢
```

where

- jobid is the batch job identification supplied by the system to the user in response to the SUBMIT statement used to place the job in the batch queue (see SUBMIT).
- userid is the user identification used to submit the batch job.
- password is the password associated with the userid.

Description: The DELJOB statement can be entered either interactively or within a batch job, to cause a specified batch job to be deleted from the batch queue.

Example:

```
DELJOB 89,JONES,XYZ␢
```

The job with jobid 89, previously placed in the batch queue via a SUBMIT statement, will be deleted from the batch queue if it is not being actively processed.

The DIR statement lists the contents of the user's file directory, or lists information on designated files in the user's directory.

```
DIR {filename1,filename2,...,filenamen}↵
```

where

- each filename designates a file or a set of files in the user's directory. If filenames are omitted, all filenames in the user's directory are listed.

  To designate a set of files, the user may default one or two of the three fields in a filename (name.extension;version). The syntax for doing this is illustrated in the following examples:

name.extension	(version defaulted)
name	(extension and version defaulted)
*.extension	(name and version defaulted)
name.*;version	(extension defaulted)
.;version	(name and extension defaulted)
*.extension;version	(name defaulted)

Example: The following is an example of the use of DIR without the filename(s):

```
DIR↵
```

```
   <FANSOME>

FILE NAME     LAST WRITE             SIZE

ASK.REL       8-DEC-72 11:34:34      199(B)
ASK.MAC       6-DEC-72 14:45:28      2816(B)
USER.DOC      5-DEC-72 22:32:17      2048(B)
```

where the SIZE entry indicates the size of the file in bytes. FANSOME is the user's userid in this example.

The EQU statement defines and names a control parameter (the first argument in the statement), and sets the control parameter equal to the integer value of an expression (the second argument).

```
EQU name,expressionb̸
```

where

- name is a user-supplied name for a control parameter. It must be made up of alphanumeric characters, must begin with a letter, and may have no more than six characters.
- expression is any expression which can be evaluated. The resulting integer value is assigned to name. An expression may be a signed value or the name of a control parameter defined in a previous EQU statement or some combination of these (see Section 6.2.5).

Description: The EQU statement is the only way in ACL to define and assign a value to a control parameter. A control parameter, once defined by EQU, may be used in subsequent ACL statements or expressions within the same interactive session, or within the same batch job.

Example 1:

```
EQU SWITCH,1b̸
```

In this EQU statement the name is SWITCH and the expression consists of the value +1. SWITCH may be a previously defined control parameter which is now given the value +1, or it may be undefined, in which case it is now defined and given the value +1. Note: Negative values are also permitted.

Example 2:

```
EQU TSTNUM,FUNNYb̸
```

Here the <u>name</u> is TSTNUM. As in Example 1, it may or may not be predefined; if it is undefined, it is defined at this time. FUNNY should be the name of a control parameter defined in a previous EQU statement, in which case its value will now be assigned to TSTNUM. In the event that FUNNY has not been previously defined, it will be evaluated as (unsigned) zero, and this value will be assigned to TSTNUM.

Example 3:

```
EQU GDNAME,DATUM-1b̸
```

Here the <u>name</u> is GDNAME, and the <u>expression</u> is DATUM-1, where DATUM is the name of a control parameter defined in a previous EQU statement. The value of DATUM minus one will be assigned to GDNAME. (If DATUM was not previously defined, it will be evaluated as zero, and the <u>expression</u> will consequently be evaluated as -1 and this value assigned to GDNAME.)

Example 4:

```
EQU VARBLE,DOIT(CHECK1,CHECK2)b̸
```

In this case the <u>name</u> is VARBLE and the <u>expression</u> is replaced by the macro-call DOIT(CHECK1,CHECK2). DOIT is the name of a previously defined macro, and CHECK1 and CHECK2 are the names of two previously defined control parameters passed to DOIT as arguments. It is assumed that the text of DOIT is an expression which, when supplied with the arguments

CHECK1 and CHECK2, will have a single integer value. If it is not, its value will either be zero or something unpredictable. In any case, a value will be returned and assigned to VARBLE.

Example 5:

```
EQU MYNAME,STATEb̸
```

Here the name is MYNAME and the expression is the special control parameter STATE. STATE is defined and maintained by the system and contains information returned by the subsystem called in the user's most recent ACL statement (see Section 6.2.4.1). In this example, the current value of STATE is assigned to MYNAME and may thus be saved for future use in an expression or in a TEST statement.

The GLYP statement is a request for a GLYPNIR program compilation and assembly. It can only be used in a batch file.

```
GLYP infilename,outfilename,{listfilename}{,ASSEMBLE=yesno}b̸
```

where

- infilename is the name of the file containing the GLYPNIR source code to be compiled.
- outfilename is the name of the file for the output.
- listfilename is the name of the file for the GLYPNIR listing of the source code. If listfilename is omitted, no listing will be produced.
- yesno is either YES or NO. If it is YES, the ASK assembler subsystem will be automatically called by the GLYP compiler subsystem and will assemble the ASK source code produced by GLYP into relocatable ASK object code in the file named outfilename. If ASSEMBLE=NO, the assembler will not be called and GLYP will output ASK source code into file outfilename. If ASSEMBLE=yesno is omitted, the default option is that the assembler will be automatically called by GLYP.

Description: The GLYP statement is a request for compilation and assembly of GLYPNIR source code. Syntax and other error messages generated by GLYPNIR and ASK will be placed in the user's primary output file (POF). If the compilation is successful, the ASK object code will be written in the file named outfilename in the user's directory. A listing of the GLYPNIR source code will be written in the file named listfilename in the user's directory.

The HELP subsystem will answer questions input by the user during an interactive session.

$\boxed{\text{HELPb̸}}$

Description: HELP answers questions about the use of the ILLIAC IV System and ACL. When the user enters a HELP statement, instructions on the use of HELP will immediately be typed out, followed by a question mark (?) which is the HELP prompt character. The user may then type in a question; HELP will type an answer and await another question.

To exit from HELP, type END followed by a carriage return.

The INQ subsystem returns information on a specified batch job, or on all batch jobs under a specified userid.

$\boxed{\text{INQ \{jobid\}}\cancel{b}}$

where

- jobid is the jobid of a particular batch job, returned by the system after the job was submitted by a SUBMIT statement. If the jobid is omitted, the system responds as described below under "Description (2)."

Description (1)

When a jobid is entered in the INQ statement, the system returns one of the following messages concerning the job specified by the jobid:

WAITING	(the job is still in the batch queue)
RUNNING	(the job is being processed)
COMPLETED	(the job is neither waiting nor running)

Caution: The response COMPLETED may be misleading, as the job may have been deleted via a DELJOB statement or the jobid may have been entered incorrectly in the INQ statement.

Description (2)

When no jobid is entered in the INQ statement, the system responds by requesting a userid. The user enters a userid followed by a carriage return, and the system returns status information on all jobs under that userid. The following is an example of the form in which this information is typed out:

```
J.ID   RC   JS   U.ID
  41   IL    W    TOM
  34   B67   R    TOM
```

Explanation: The columns show the jobid (J.ID), the runcode (RC), the job status (JS), and the userid. The runcode is IL (ILLIAC IV), B67 (B6700), BT (both ILLIAC IV and B6700), or NE (neither ILLIAC IV nor B6700). The job status is W (waiting) or R (running).

The LIB statement is used to create a named library of relocatable ASK object code files. This library can then be used by the link editor (under control of the SEARCH statement in the link editor call – see LINKED) to link to user programs contained in the library. The output of the LIB subsystem is a file containing tables referencing the code files that make up the library and the entry points in those files.

```
LIB libname,filename1,filename2,...,filenamenƀ
```

where

- libname is a user-supplied filename to be assigned to the library file. If no extension is supplied, the default extension will be LIB.
- filenames are the names of files of relocatable ASK object code to be included in the library (i.e., referenced in the library file). If extensions are omitted from any of these filenames, the default extension in each case will be REL.

Description: The LIB subsystem builds a file containing two tables. The first table is a list of the file-names supplied in the LIB statement, and the second is a table of all the entry-point labels in these files. (Entry points are labels declared to the assembler as entry points.) The entry-point table can be searched by the link editor under control of the SEARCH statement (see LINKED).

The addition or deletion of any entry point in any of the files referenced in the library file requires the library to be explicitly amended; this is done by re-creating the library file with a new LIB statement.

Currently, a library file may contain any number of filenames and up to 54,000 entry points.

The LINKED statement calls the link editor subsystem, which converts relocatable ASK object code into an ILLIAC IV SAVE (ISV) file. The ISV file is subsequently input to the RUN subsystem for execution on the ILLIAC IV or to SSK for simulation. LINKED requires control statements to specify its operation.

```
LINKED {infilename},outfilename{,OPTIONS=optionstring}␢
```

where

- infilename is the name of a file containing LINKED control statements for input to the link editor. If infilename is omitted, control statements are assumed to be in-line directly following the LINKED statement. If infilename is entered without an extension, the default extension LNK is used.
- outfilename is the name of the file that will contain the link-edited ILLIAC IV image file. This image file is referred to as an ILLIAC IV SAVE file. If outfilename is entered without an extension, the default extension ISV is used.
- optionstring is a string of letters each of which controls an optional feature of the link editor. Any or all of these letters may be omitted. Currently, the following options are available:

X	Do not produce an output file if any errors are detected.
N	Do not produce a map of the image file contents.
L	Do not search the system library for external symbols.
M	Do not produce a disk map listing for the image file.

Option letters may be strung together in any order, and it is permissible to repeat them within the string.

Description: The function of LINKED is to convert relocatable ASK object-code programs to absolute ILLIAC IV (Array Memory) addresses, to resolve linkages, and to create an ILLIAC IV SAVE (ISV) file. The output from LINKED may be input directly to SSK for simulation.

Each call to LINKED results in a single output ISV file containing an ILLIAC IV Array Memory image and additional information to set the initial machine state. The ISV file will be subsequently loaded by the RUN subsystem or simulated by SSK. LINKED will assign a standard address to the origin of the user's program; the user can assign a different origin address with an optional control statement called SET. (Control statements are described individually in detail below.) The user may also specify an initial entry point other than the first entry point of the first relocatable file specified to LINKED, using an optional control statement called ENTRY.

Control Statements for LINKED Subsystem:

The statements used as control input to the LINKED subsystem are called INCLDE, SET, ENTRY, and END. The first, INCLDE, is used to specify the files of relocatable ASK object code to be included in the image file.

The INCLDE Statement:

The INCLDE statement has the following format:

```
INCLDE filename1,filename2,...,filenamen⁊
```

The <u>filenames</u> are the names of files of relocatable ASK object code to be processed by the link editor and included in the output image file. If a <u>file-name</u> is entered without an <u>extension</u>, the default <u>extension</u> REL will be used. The files will be processed by the link editor in the order given in the INCLDE statement. The format

```
INCLDE filename1,filename2ƀ
INCLDE filename3,filename4ƀ
```

is exactly equivalent to

```
INCLDE filename1,filename2,filename3,filename4ƀ
```

The SET Statement:

The SET statement is used optionally to specify an address for the origin of the user's program. It may be entered anywhere in the sequence of statements for the LINKED subsystem. If more than one SET statement is used in the sequence of statements following the LINKED statement, the last SET statement prevails. If no SET statement is used, a standard address will be used. The SET statement has the following format:

```
SET addressƀ
```

where address is a syllable address in Array Memory. A "syllable" is a 32-bit half-word; syllable addresses are numbered sequentially across rows, starting from syllable 0 at the beginning of Row 0 and ending with syllable 262,144 at the end of Row 2047.

The address may be given in either decimal or octal notation. To indicate octal notation, a "pounds" sign (#) is used as a prefix; otherwise, decimal notation is assumed. For example, there are 128 = #200 syllables on each row.

The address should specify a syllable that is the first syllable on a row. Observe that syllables #0, #200, #400, #600, #1000, ..., #777,600 are the syllable addresses at the beginning of rows, since there are #200 syllables on each row. If the address does not lie at the beginning of a row, the assigned origin address will be the next syllable address that does lie at the beginning of a row — for example, if the address given is #236, the assigned origin address will be #400.

The ENTRY Statement:

The ENTRY statement is used optionally to specify an initial program entry point other than the first entry point in the first file included in the INCLDE statement. The ENTRY statement may be used anywhere in the sequence of statements for LINKED. If more than one ENTRY statement is used, the last one prevails. If no ENTRY statement is used, the entry point of the program will be the first entry point in the root segment. The ENTRY statement has the format

```
ENTRY label␢
```

where label is the label of an instruction in the user's relocatable ASK code and has been declared as an entry point at assembly time.

The END Statement:

The END statement is used at the end of the sequence of statements for the LINKED subsystem. It may not be omitted. The END statement has the format

```
END␢
```

The END statement has the effect of terminating the call to the LINKED subsystem; LINKED will then perform all the necessary operations to create the image file as specified in the preceding control statements.

The LOGIN statement is not an ACL statement but is included here for reasons that will be obvious. It is the means for starting an interactive ACL session on the ILLIAC IV System, and _must_ be the user's first statement upon establishing a Network connection to the ILLIAC Host.

$\boxed{\texttt{LOGIN userid password account}\flat}$

where

- _userid_ is the user's identification, administratively established beforehand.
- _password_ is the _password_ associated with the _userid_ and will not be echoed at the user's terminal when entered.
- _account_ is a string of digits chosen by the user (in the current implementation it is not used by the System).

Note: Unlike ACL syntax, the syntax of the LOGIN statement calls for the arguments to be separated by spaces, not commas.

Description: On completion of LOGIN, the ILLIAC IV System responds with a (!) prompt signal to indicate that it is ready to accept an ACL statement.

The LOGOUT statement causes a normal termination of an ACL batch job or interactive session. LOGOUT does not disconnect the user terminal from the ILLIAC Host.

```
LOGOUTƀ
```

Description: In the interactive mode LOGOUT causes normal termination of the session. The status of any batch jobs submitted (see SUBMIT) by the user during this interactive session will be unaffected even though the user has logged out.

If the LOGOUT statement is included in an ACL statement file, it is equivalent to an EOF. It may be used in this sense to create alternative exit points in a batch job.

Caution: In the interactive mode, LOGOUT terminates an ACL session but does not cause a disconnect. The user then has two options:

(1) Disconnect from the ILLIAC Host

(e.g., TELNET, ↑Z)

(2) Resume ACL interaction

(e.g., TELNET, ↑C followed by LOGIN)

LOGOUT has this effect whether it is entered from the terminal or in a file of ACL statements processed via an ALT statement.

The MACRO statement causes the sequence of text that follows it (enclosed in square brackets []) to be defined as an ACL macro. Subsequent to the macro definition, the macro may be called in ACL by its name.

```
MACRO name{,(arg1,arg2,...,argn)}ƀ
```

where

- name is the user-supplied name by which the macro will be called. The name may be any string of alphanumeric characters.
- (arg1,arg2,...,argn) is the formal argument list of the macro. The macro subsystem imposes no restrictions on the construction of formal arguments other than the general caution against arbitrary use of special characters such as the semicolon. The parentheses enclosing the formal argument list are required, as is the comma separating the argument list from the name.

Text of a Macro: The text of a macro is a character string, an expression, or one or more ACL statements in legal combination and is enclosed in square brackets.

Description: Macro processing occurs in two steps:

Step 1 is the process of defining the bracketed sequence of text as an ACL macro. After this, the macro name is "known" to the system and may be subsequently called.

A macro definition is global with respect to the job within which it is defined. Here "job" means (a) a user's interactive session, not including any batch job that he submits during that session; or (b) a batch job, including secondary sequences of ACL statements not contained in the PIF (files branched to by ALT statements, for example). Within a job, a macro defined in one file may be called in another, assuming that the definition has been processed before the call is processed.

Step 2 is the macro expansion, which occurs (after the macro definition) when the defined macro is called. A macro is called either when it is invoked directly as if it were an ACL statement or when the macro name is referenced as an argument in another ACL statement.

An example of invoking a macro:

```
EQU JAZZ,JAZZ+1ƀ
EQU BIT5,[CHECK←5]\35ƀ
:
SAFETY(JAZZ,BIT5)ƀ
:
```

where SAFETY is the name of a previously defined macro having two formal arguments; JAZZ and BIT5 are control parameters passed to SAFETY as actual arguments. JAZZ and BIT5 will be inserted in the text of SAFETY in place of the formal arguments.

An example of referencing a macro:

```
EQU PARAM,CALC(STRNUM,5,UNK)ƀ
```

where CALC is the name of a previously defined macro having three formal arguments and STRNUM and UNK are previously defined control parameters passed to CALC as the first and third actual arguments; the value 5 is passed as the second actual argument.

(This distinction between invoking and referencing a macro is not a rule of construction imposed by the MACRO subsystem. The terminology is introduced here to explain the effects of this difference in usage.)

A defined macro may be referenced in place of any ACL statement argument, but may not appear in an expression. A macro may be called from another macro.

The expansion process of a macro is an in-line string insertion of the text contained in the defined macro (with actual arguments inserted in place of formal arguments). When a macro is referenced, the macro text is inserted in-line in the referencing ACL statement. After this insertion of the text, the MACRO subsystem processing is complete.

A macro which is to be invoked may contain any combination of ACL statements. A macro which is to be referenced may contain either a character string or an expression. A referenced macro which yields an expression is analogous to a FORTRAN function call.

The text of a macro is not checked for violation of ACL construction rules by the macro subsystem. For this reason, the content of a macro definition should be viewed as text. After the macro is expanded, normal ACL rules of construction apply.

A macro definition must include a macro name, and may include a formal argument list, (arg1,arg2,..., argn), enclosed in parentheses. A macro defined with a formal argument list may be called (i.e., invoked or referenced) with or without the arguments included in the calling sequence. Arguments which are omitted in a calling sequence must be delimited with the "," so that a correct argument sequence correspondence is established. When arguments are omitted in the calling sequence, a "null" character is inserted for the omitted parameter in the text of the macro expansion. A "null" character is <u>not</u> equal to zero. Unless caution is exercised, a formal argument omitted in a calling sequence may cause a subsequent syntactic error. See Example 4 below.

Examples of macro usage:

Example 1:

The macro definition:

```
MACRO NEWNAMEƀ
[PROC.JDATA]ƀ
```

when referenced in:

```
DIR NEWNAMEƀ
```

results in the processing of the ACL statement:

```
DIR PROC.JDATAƀ
```

This is an example of a macro used to insert a character string in a referencing ACL statement. It illustrates the explanation above, that the macro expansion process is simply a string insertion of the text contained in the macro definition.

Note also that a macro definition does not have to include a formal argument list.

Example 2: The macro definition:

```
MACRO CHECK,(THIRD)b̸
[EQU THIRD,STATE←3b̸
EQU THIRD,THIRD\35]b̸
```

when invoked in

```
CHECK(TEST1)b̸
```

sets the value of TEST1 to the value of the Bit 3 of the STATE parameter. TEST1 is now defined and a value assigned to it, and it may be subsequently tested with the TEST statement.

Assuming the ACL statements immediately before and after the macro call are STATEMENT1 and STATEMENT2, the macro invocation above would result in:

```
STATEMENT1b̸
EQU TEST1,STATE←3b̸    ⎫ macro expansion
                       ⎬       and
EQU TEST1,TEST1\35b̸   ⎭ text insertion.
STATEMENT2b̸
```

Example 3: The macro definition:

```
MACRO DEFILE,(FILENAME)b̸
[DEL FILENAMEb̸
DIRb̸
EQU CHECK,1]b̸
```

when invoked in:

```
DEFILE(MYFILE,SOURCE;1)b̸
```

causes the file named "MYFILE.SOURCE;1" to be deleted, the user directory to be listed, and a control parameter CHECK set to a positive 1.

The CHECK control parameter may be subsequently used in an ACL statement. For example,

```
...
TEST CHECK,(NEGCASE,ZEROCASE,POSCASE)b̸
```

The subsequent use of CHECK in this example illustrates the fact that control parameters defined in a macro expansion are global definitions even when not in the argument list of the macro.

Example 4: The macro definition:

```
MACRO EVALU,(CC)b̸
[AA+[BB*CC]]b̸
```

when referenced in the sequence:

```
EQU AA,1b̸
EQU BB,0b̸
...
EQU DD,EVALU(5)b̸
```

causes DD to be assigned the value +1. This is an example of the macro capability being used in a manner analogous to a FORTRAN function. Since the macro expansion process is a string insertion, the expansion results in the ACL statement:

```
EQU DD,AA+[BB*5]b̸
```

which causes the expression AA+[BB*5] to be evaluated (as +1) and the value assigned to DD. From this example it should be noted that when a macro call is referenced in an EQU (or TEST) statement, its text must be a legal expression — i.e., a legal argument of the statement. If the ACL sequence in the example above were

```
EQU AA,1b̸
EQU CC,0b̸
EQU BB,5b̸
...
EQU DD,EVALUb̸
```

with the actual argument in the macro call omitted, the macro expansion of EVALU would be successful, but a syntax error would occur in the evaluation of the resultant statement,

```
EQU DD,AA+[BB*]b̸
```

because "BB*" is not a valid expression; hence "AA+[BB*]" is not a valid expression, and thus it is not a valid second argument for the EQU statement.

The MAP subsystem accepts a user-defined ILLIAC IV Disk Memory (I4DM) layout and creates a file of allocation tables for subsequent input to the ALLOC subsystem. The disk memory layout is specified by means of control statements.

```
MAP {infilename}{,mapfilename}ƀ
```

where

- infilename is the name of a file containing control statements for the MAP subsystem. If infilename is omitted, MAP subsystem statements are assumed to be in-line following the MAP statement.
- mapfilename is the name of the file to be used by the MAP subsystem for output of the disk memory allocation tables. If mapfilename is omitted, no allocation tables are created. Normally this option would be used only if TIME or PRINT control statements were included (see below) and the user wanted to produce MAP output listings.

Reference: See Section 5 of this guide for a detailed discussion of how to lay out the ILLIAC IV Disk Memory.

Discussion: The MAP subsystem accepts and processes control statements entered directly (in-line) or in a file. These statements describe an I4DM layout which is used by MAP to create a file of allocation tables; these tables are in turn used by the ALLOC subsystem to assign I4DM space in preparation for execution of an ILLIAC IV program.

The user's description of an I4DM layout, as expressed in control statements to MAP, is in terms of the logical disk, which is a logical model of the physical I4DM system presented to the user by the disk utility software. The following is a condensed discussion of the logical disk and page-identification conventions associated with it; for a full discussion, see Section 5.2.

The Logical Disk:

The smallest collection of data on the I4DM system that is logically addressable is a page. A page of data is a contiguous collection of 1024 (64-bit) or 2048 (32-bit) ILLIAC IV words. The I4DM layout is structured at a page level.

Contiguous pages are grouped in sequential sets of 300 into bands. A total of 52 bands, or 15,600 pages, are available on the I4DM.

There is a time delay (caused by electronic switching) equivalent to two pages of rotational time delay when (1) accessing pages across a band boundary (any one band to any other band), or (2) between any two successive data transfer requests. Note that all bands are "equidistant" from each other in terms of time.

The page and band layout is a logical structure of disk memory space. Pages and bands are uniquely identified to the MAP subsystem relative to the position of Band 0, Page 0. See Section 5 for further explanation.

Page Identification:

All band or page assignments made in a layout are denoted to the MAP subsystem relative to Page 0, Band 0 of the logical disk. Bands are identified in sequence from 0 to 51. Notationally, Band 52 is the same as Band 0 (i.e., band numbers are considered modulo 52). Each band has 300 pages numbered from 0 to 299. Page 299 of any band is contiguous to Page 0 of the _same_ band; page numbers are considered modulo 300.

As implied by this structure, the I4DM may be conceptually viewed as a cylinder, with Pages 0 to 299 running around the cylinder in 52 bands and with Page 299 contiguous to Page 0 in any one of the bands. Band 51 is notationally adjacent to Band 0 (recalling that there is a two-page time delay to access from Band 0 to Band 51 just as there is between _any_ two bands). See Figure MAP-1.

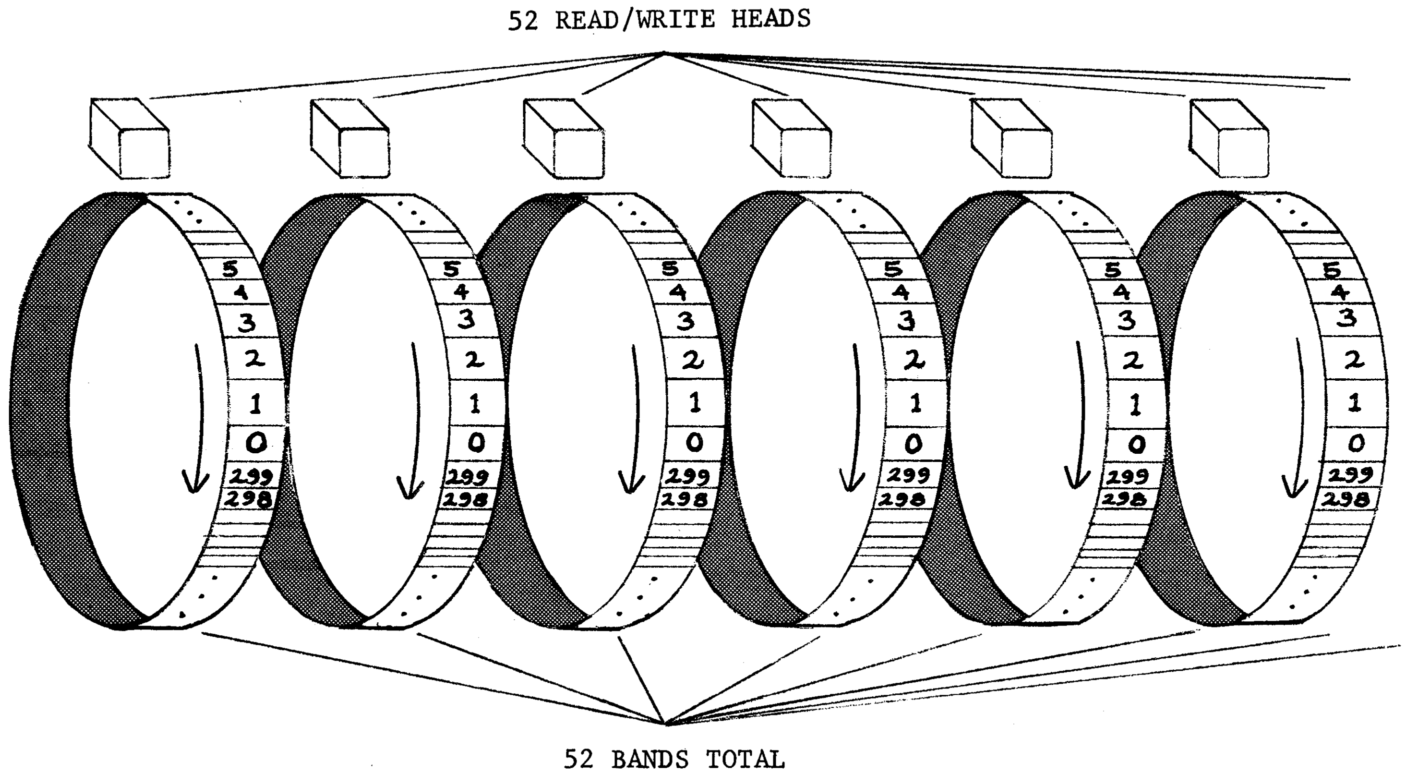


Figure MAP 1. Logical Disk Structure, Showing 6 of the 52 Bands

Control Statements for the MAP Subsystem:

The MAP subsystem statements are FORMAT, PRINT, TIME, and END. A complete call to the MAP subsystem consists of a MAP statement, followed by at least one FORMAT statement. PRINT and TIME statements are optional. The call is terminated by the END statement. The control statements are discussed individually below.

The FORMAT Statement:

The FORMAT statement is used to specify the arrangement on the logical disk of one <u>area</u>. An area is a named collection of user-specified pages on the logical disk. The FORMAT statement associated with the area identifies the pages in the area, specifies the arrangement of these pages on the logical disk, and assigns a name (the <u>areaname</u>) to the collection of pages.

The FORMAT statement has the following format:

```
FORMAT areaname,(formatspec)ƀ
```

where

<u>areaname</u> is a user-supplied name to be assigned to the area described in this FORMAT statement (up to six alphanumeric characters, with a letter for the first character).

<u>formatspec</u> is a parenthesized sequence of operators that identify logical disk pages which are to be assigned to the area. <u>formatspec</u> defines the relative arrangement on the logical disk of these pages. <u>formatspec</u> is discussed in detail in the following section.

The formatspec:

The MAP subsystem maintains a "current page position pointer," used in the processing of FORMAT statements, that always points to some logical disk band and some page within the band, in accordance with the logical disk of Figure MAP-1. For each FORMAT statement, this pointer is initially set to Page 0, Band 0. The operators that make up the formatspec move the pointer in various ways. The operators will be considered in detail, starting with the operator used to reserve pages in the area.

The ±nP Operator:

This operator has the form ±nP, where n is a signed integer. The n in this operator may be omitted if it is 1, and the sign may be omitted if it is positive.

The effect of ±nP is to reserve a set of n consecutive pages, starting with the current pointer position as the first page of the set and proceeding forward or backward on the logical disk (i.e., with or against the direction of physical rotation), depending on the sign of the operator. The pointer is left pointing to the next page on the logical disk, following the last page of the reserved set.

At the beginning of each formatspec, the pointer position is Band 0, Page 0. Thus the simple formatspec

(128P)

would specify an area consisting of 128 contiguous pages starting at Band 0, Page 0 and extending to Band 0, Page 127. The pages are contiguous because they are all within the same band.

The ±nS Operator:

The n in this operator may be omitted if it is 1 and the sign may be omitted if it is positive. The ±nS operator moves the current page position pointer n pages forward or backward, depending on the sign, and has no other effect. The formatspec

(32S,64P)

would first move the pointer from its initial position at Page 0, Band 0 to Page 32; the 64P operator would then reserve Pages 32-95. The resulting area would consist of 64 contiguous pages starting at Page 32, Band 0 of the logical disk. The pointer would be left at Page 96, Band 0.

Another use of the ±nS operator is seen in the following example:

(32P,2S,32P)

Here 32 contiguous pages are reserved as part of the area, then the pointer is moved two pages forward, and another 32 contiguous pages are reserved.

Action of ±nP and ±nS Operators at a Band Boundary:

Before discussing other operators, one more characteristic of ±nP and ±nS operators will be described. When either of these operators moves the pointer past Page 299 of any given band, the "next" page (for the purposes of the operator) will be Page 0 of the "next" band. The formatspec operators are implemented in this fashion to facilitate the handling of large data arrays.

The "next" band is the next band in numerical sequence counting from Band 0 through Band 51, with Band 0 following Band 51. This sequence is a notational property of the formatspec operators; as stated before, the bands of the logical disk are equidistant in terms of access time, and thus the sequence does not imply a physical relationship.

When a negative operator moves the pointer past Page 0 of any band, the "next" page will be Page 299 of the "previous" band.

For example, if the current pointer position is Page 290 of Band 13 and a 50P operator is encountered, Pages 290-299 of Band 13 and Pages 0-39 of Band 14 will be reserved, for a total of 50 pages. If the pointer position is Page 40 of Band 13 and a -50S operator is encountered, the pointer will be moved to Page 289 of Band 12.

The ±nB, ±nL, and ±nR Operators:

The n in these operators may be omitted if it is 1 and the sign may be omitted if it is positive. These operators, like the ±nS operator, have no effect except to move the pointer; they provide additional convenience and flexibility by moving it in different ways.

The ±nB operator moves the pointer n bands from its current position, maintaining the pointer at the same page number within the new band. For example, if the current position is Page 234, Band 16 and a -4B operator is encountered, the resulting pointer position will be Page 234, Band 12. If a ±nB operator moves the pointer "past" Band 51 in a positive direction, it moves from Band 51 to Band 0, then to Band 1, etc. in sequence. If the pointer is moved in a negative direction "past" Band 0, it moves from Band 0 to Band 51, then to Band 50, etc. in sequence.

The ±nL operator resets the pointer to Page 0, Band 0, and then moves it to the nth page of the logical disk counting pages consecutively from Page 0 (Band 0). There is a total of 15,600 pages on the disk, and for purposes of the ±nL operator they form a continuous sequence, where Page 0, Band 0 is the 0th page on the disk and Page 299, Band 51 is the 15,599th page on the disk. Page 15,599 is "followed" in this sequence by Page 0. The n in a ±nL operator is taken modulo 15,600. Note that the pointer position resulting from a ±nL operator is independent of the previous position.

For example, the formatspec

```
(500L,10P)
```

moves the current page position pointer to Page 500 (i.e., Page 200 of Band 1) and then reserves ten pages, Pages 200 to 209 of Band 1. This sequence is independent of what precedes the L operator in the formatspec. For example, in the formatspec

```
(64P,2S,64P,500L,10P)
```

the 500L, 10P operator sequence again reserves Pages 200 to 209 of Band 1.

The ±nR operator moves the pointer to the nth page within the current band, taking n modulo 300 and counting from Page 0 of the current band. The nth page within the band is counted from Page 0 of the band, regardless of the pointer position within the band before the operator is encountered. For example, if the pointer is in Band 10 and Page 100 when a 5R operator is encountered, the resulting position will be Page 5, Band 10. For purposes of this operator, the pages within a band are considered to "wrap around" from Page 299 to Page 0; thus a -5R operator, in the above example, would move the pointer to Page 295 of Band 10, and a 310R operator would move the pointer to Page 10 of Band 10.

Combinations of Operators:

As noted above, operators are separated by commas and are interpreted and processed in sequence. Two or more operators may be enclosed in parentheses prefixed by a signed integer ±n; the parenthesized sequence of operators is repeated n times. For example,

```
3(2S,50P)
```

is exactly equivalent to 2S,50P,2S,50P,2S,50P. If the integer is negative, the signs of all operators in the parenthesized sequence are changed, e.g.,

```
-2(2S,-50P)
```

is exactly equivalent to -2S,+50P,-2S,+50P. Parenthesized sequences may be nested as in

```
2(2S,3(20P,2S))
```

which is exactly equivalent to 2S,3(20P,2S),2S,3(20P,2S) or 2S,20P,2S,20P,2S,20P,2S,2S,20P,2S,20P,2S,20P,2S.

The Continuation Character in FORMAT Statements:

In a FORMAT statement, the <u>formatspec</u> may be very lengthy. The semicolon may be used as a continuation character within the <u>formatspec</u>. The semicolon should immediately follow a comma and should be followed by a carriage return.

<u>Example</u>

```
FORMAT INDATA,(3P,2S,4P,4S,6P,2S,20(S,P),2S,P,10(S,P),;ƀ
S,15(S,P),2S,10P)ƀ
```

Summary of the FORMAT Statement:

The FORMAT statement is used in a MAP subsystem call to name and describe the arrangement on the logical disk of one area. Within one MAP subsystem call, all FORMAT statements use a common reference point (Page 0, Band 0 of the logical disk) for the area descriptions they contain. The description of an area is called a <u>formatspec</u> and is composed of operators. One operator (±nP) is used to reserve n consecutive pages in the area; other operators move a logical current page position pointer that is used to locate on the logical disk the sets of pages laid out by ±nP operators. The areas described in a FORMAT statement will be assigned space on the I4DM by the ALLOC subsystem.

Important Note: One aspect of the FORMAT statement is not discussed in this section: the sequence in which pages are reserved in the formatspec. This sequence is significant, and two formatspecs are not equivalent if they reserve the same logical disk pages but in a different sequence. See Section 5.4 for a complete discussion.

The Map Subsystem List Outputs: There are two operators that can be used in a FORMAT statement which have not yet been described. These are:

X — which means mark the current page for printing; marked pages are recognized by the PRINT statement (see below).

T — which means mark the current page for timing; marked pages are recognized by the TIME statement (see below).

"Current" in the above definitions refers to the current position of the page pointer.

Note that unlike the other operators, X and T are not prefixed with a signed integer. The effect of X or T is to mark the current page for printing or timing, without moving the pointer. Thus the formatspec

(X,20P,2S,X,30P)

reserves two sets of 20 and 30 pages respectively, with a two-page gap between them, and the first page of each set is marked for printing. To mark all reserved pages for printing, in the same layout, one would write

(20(X,P),2S,30(X,P))

The construction X,T or T,X marks the current page for both timing and printing. The following formatspec again reserves two sets of 20 and 30 pages respectively, with a two-page gap between them, and marks the first page of each set for timing and all reserved pages for printing:

(T,20(X,P),2S,T,30(X,P))

Note: It is syntactically permissible to prefix an X or T operator with a signed integer, e.g., 5T. However, the signed integer in this case is ignored, and 5T is exactly equivalent to T.

The PRINT Statement:

The PRINT statement has the following format:

```
PRINT areaname1,areaname2,...,areanamen$\not b$
```

where

areaname(s) have been previously specified with FORMAT statements in this MAP call.

The PRINT statement causes MAP to produce a diagram representing the logical disk, and indicating all pages marked for printing in preceding FORMAT statements. The diagram consists of 52 columns and 100 rows; each column represents a band (with Band 0 at the left), and each row entry within a column represents three contiguous pages within the band (with Pages 0, 1, 2 at the top of each column). The three pages are internally represented by a pattern of three bits — 100 if the first of the three pages is marked for printing, 110 if the first and second are marked, etc. On the PRINT diagram, each entry consists of the corresponding octal number, i.e., 0 if none of the three pages is marked, 3 if the second and third are marked, etc.

The TIME Statement:

The TIME statement has the following format:

```
TIME areaname1,areaname2,...,areanameny
```

where

<u>areaname(s)</u> have been previously specified with FORMAT statements in this MAP call.

The TIME statement writes out the I4DM rotational time delay between pages marked for timing in preceding FORMAT statements. Time delays will be computed between pages in the sequence in which they occur on the logical disk, and not in the sequence of the area virtual map (see Section 5.4 for a discussion of the I4DM area as a virtual map).

The END Statement:

The END statement has the following format:

```
ENDy
```

An END statement is required as the last control statement to the MAP subsystem. Upon recognizing the END statement, MAP will process all the other control statements in the sequence and will produce a file of allocation tables, TIME output, and/or PRINT output, as previously specified by the user.

Each user is assigned a certain amount of active file space, which is used for active file storage. The MAXAFS statement causes the system to type out the maximum number of pages currently assigned to the user for active file storage.

```
MAXAFS␢
```

Description:	Type the maximum number of active file storage pages assigned to the user.

The MOVE statement is used to transfer the contents of an active file in the user's directory to an area on the I4DM, or to transfer the contents of an I4DM area to a file in the user's directory. MOVE can only be used in a batch job.

```
MOVE source,destinationɲ
```

where

- source designates the file or I4DM area to be copied from.
- destination designates the file or I4DM area to be copied to.

One of the two arguments must designate a file in the user's directory, specified by a filename; the other argument must designate an I4DM area which the user has described in a MAP subsystem call and which has had space assigned to it on the I4DM by the ALLOC subsystem. An I4DM area is specified as source or destination as follows:

```
I4DM:areaname
```

where areaname is the name used in a FORMAT statement within the MAP subsystem call. The characters "I4DM:" must be entered literally in the MOVE statement.

Description: MOVE transfers the contents of the source (file or I4DM area) to the destination (file or I4DM area). If the transfer is successful, a message is returned to the user indicating the number of bytes transferred. If unsuccessful, MOVE returns an appropriate diagnostic message.

Note: An I4DM area has a fixed number of pages assigned to it. Therefore, if destination is an I4DM area and the source file contains more pages than are assigned to the area, the area will contain a truncated copy of the file after MOVE is accessed.

Examples:

```
MOVE DATA.INPUT,I4DM:INDATA␢
```

Transfer the contents of file DATA.INPUT to area INDATA.

```
MOVE I4DM:OUTDAT,DATA.OUTPUT␢
```

Transfer the contents of area OUTDAT to file DATA.OUTPUT.

The NEWS subsystem types out news about ACL subsystems.

NEWSƀ

Description:

When the NEWS statement is entered, the NEWS subsystem prompts the user by typing "NEWS ABOUT:" and then awaits further input. The user may then type in the name of a topic (such as an ACL subsystem) followed by a carriage return. NEWS will then type out the news relating to the specified topic. One of the available topics is "GENERAL", which includes instructions on the use of NEWS and a list of other available topics.

The user may also respond with nothing but a carriage return, which causes all the news to be typed out.

After typing out news on whatever topic the user has selected, NEWS again prompts the user by typing "NEWS ABOUT:". The user may then respond with another topic, or exit from NEWS by typing "ENDƀ".

While NEWS is typing out, the user may interrupt by striking control O (strike "CONTROL" and "O" keys simultaneously). NEWS will stop typing (not necessarily immediately) and prompt for a new topic.

The NOTIFY statement routes a message to the system operating staff.

```
NOTIFY "message"␢
```

where

- message is a character string of any length, enclosed in quotes. Carriage returns may be used within the message.

Description: NOTIFY displays a user's message to the system operating staff. If the message requires action, the staff can communicate with the user by returning a message to the user's terminal, or by leaving a message in a file in the user's directory. The NOTIFY statement may only be input interactively. Messages sent with a NOTIFY statement will have a header added automatically which identifies the user and provides the time and date.

If the message is a report of a problem in the system, the message will be directed to the appropriate system operating staff member for information and action. For messages that report system problems, operating procedures require the staff to acknowledge the user's message.

System Response: Successful execution of the NOTIFY subsystem causes the following message to be typed on the user's terminal:

```
LINK FROM userid, TTY number↲
ACL NOTIFY MESSAGE TO OPERATOR date time↲
message↲
BREAK↲
```

where <u>userid</u> and <u>number</u> are the user's <u>userid</u> and teletype number, and <u>message</u> is a repeat of the user's <u>message</u>.

If the user's <u>message</u> is not successfully sent, the system will type:

```
CONSOLE TTY BUSY OPERATOR ENGAGED AT THIS TIME↲
```

If no message is included in the NOTIFY statement, the system will type:

```
MESSAGE MAY NOT BE DEFAULTED↲
```

Example:

```
!NOTIFY "MY JOB DISAPPEARED FROM THE BATCH QUEUE↲
FOR NO APPARENT REASON"↲
LINK FROM FANSOME, TTY 25↲
ACL NOTIFY MESSAGE TO OPERATOR 29-FEB-73 9:48:31↲
"MY JOB DISAPPEARED FROM THE BATCH QUEUE↲
FOR NO APPARENT REASON"↲
BREAK↲
!
```

The OUTPUT statement is used to specify full or half duplex terminal operation and the output line width on the user's terminal.

```
OUTPUT {LINE=mode}{,WIDTH=linewidth}b̸
```

where

- mode is either FULLDUPLEX or HALFDUPLEX and specifies the communication mode between the system and the user's terminal. (Actually, only the first character of mode is scanned, so mode may be any arbitrary word starting with F or H.) The default mode is FULLDUPLEX. The mode must be preceded by the string "LINE=".
- linewidth is the desired output line width on the user's terminal in characters, e.g., WIDTH=50 for a 50-character line width. The default line width is 72 characters. WIDTH may not be set to less than 8 characters. The linewidth must be preceded by the string "WIDTH=".

Description: When the user logs in, the ACL executive assumes full duplex operation with a 72-character line width. The OUTPUT statement is used to change to half duplex mode and/or to change the line width.

Note: Both arguments (LINE=mode and WIDTH=linewidth) are "keyword arguments." They need not be specified in the order shown above, i.e., their positions in the OUTPUT statement may be reversed. If either of the two arguments is omitted, no comma is needed. The only rule is that if both arguments are used, they must be separated by a comma.

Examples:

```
OUTPUT LINE=HALFDUPLEX,WIDTH=64b̸
```

```
OUTPUT WIDTH=50b̸
```

```
OUTPUT LINE=Hb̸
```

```
OUTPUT WIDTH=70,LINE=Hb̸
```

The RENAME statement changes the filename of a user's file in the ILLIAC IV System.

```
RENAME oldfilename,newfilenameƀ
```

where

- oldfilename is the name of the file to be renamed.
- newfilename is the new name for the file.

Description: RENAME replaces the oldfilename (i.e., name.extension;version) in the user's directory with newfilename (i.e., name.extension;version). The standard ACL file naming conventions apply to the RENAME statement. Once the RENAME statement is processed, the oldfilename no longer exists and will not be recognized by the system. If version number is not specified with newfilename, it will be the highest existing version number.

Example:

```
RENAME ALPHA.NDF,BETA.RDƀ
```

The highest numbered version of file ALPHA will be renamed.

The RUN subsystem is used to load an ILLIAC IV program and initiate its execution. The RUN statement can only be used in a batch file.

```
RUN infilename,{DMPFIL=dumpfilename}{,MAXTIM=time}ƀ
```

where

- <u>infilename</u> is an ILLIAC IV SAVE (ISV) file produced by the LINKED subsystem. If the <u>infilename</u> is entered without an <u>extension</u>, the default <u>extension</u> ISV will be used.
- <u>dumpfilename</u> is the name of a file to contain an image of the contents of Array Memory upon any termination of the ILLIAC IV program. If DMPFIL=<u>dumpfilename</u> is omitted, no dump file is created.
- <u>time</u> is the maximum number of seconds the user's ILLIAC IV program will be allowed to run. If <u>time</u> is exceeded, the program will be terminated; this will cause a dump file to be created if a <u>dumpfilename</u> has been specified. If MAXTIM=<u>time</u> is omitted, no maximum time is set.

Description: The RUN subsystem goes through four steps:

(1) Allocate space for the program in ILLIAC IV Disk Memory (two bands are required for this purpose).
(2) Transfer the input ISV file to ILLIAC IV Disk Memory.
(3) Transfer the information from disk memory to Array Memory and the ILLIAC IV Processor.
(4) Initiate execution.

Step (1) will fail if the user has not left at least two bands clear in disk memory when laying out disk areas with the MAP subsystem. If any of Steps (1) through (3) fail, Step (4) will not occur.

Program execution will begin at one of the following locations within the user's program:

(1) If a starting location was specified with an ENTRY statement to LINKED (see LINKED), execution will begin at the specified location.

(2) If no ENTRY statement was used, execution will start at the first entry point (entry points are labels declared to the ASK assembler as entry points).

(3) If no ENTRY statement was used and no entry points have been declared, execution will start at the origin of the user's program.

After a RUN statement is processed, the next statement in the job will not be processed until execution terminates on the ILLIAC IV. Execution may terminate under program control, or because the _time_ set by the user is exceeded, or for other reasons. In any case, the RUN subsystem will produce a dump file upon termination if DMPFIL=_dump-filename_ has been specified.

The SEND statement causes a message to be sent (as a file) to one or more other users who have file directories in the ILLIAC IV System.

$$\boxed{\text{SEND}\cancel{b}}$$

System Response: When the SEND statement has been entered, the SEND subsystem prompts the user with "TYPE LIST OF USERS." The user may then type in any number of directory-names, separated by commas and terminated by a carriage return. (Note: The directoryname for a given user is that user's userid.) Next the system prompts the user to type in a message, which will be sent to each of the specified file directories. The message may be edited while it is being typed in (see below), and is terminated by a control Z (↑Z).

Description: In each addressee's file directory, the SEND subsystem creates a file called MESSAGE.TXT, containing the message with a header identifying the sender and giving the time and date. In addition, a temporary file called MESSAGE.COPY is created in the sender's file directory. Each addressee will receive a herald message on his terminal the next time he logs in, notifying him that he has a message. He may then use DED to type out the contents of file MESSAGE.TXT.

If the file MESSAGE.TXT already exists, the new message is appended to it.

Control Characters: The following control characters may be used for editing during entry of the message:

↑A	Delete the last input character.
↑Q	Delete back to the beginning of the line.
↑R	Retype the current line. (User may then continue typing line; useful when ↑A has been used several times and the line is hard to read.)
↑X	Delete back to the beginning of the message.
↑Z	Terminate message and exit from SEND.
↑B	Insert the contents of a file into the message. When ↑B is entered, the system prompts the user to supply a filename. The filename is terminated with a carriage return or line feed. The user may then continue typing in the remainder of the message, if any, or terminate the message with ↑Z.

The SSK statement is a request to simulate the execution of an ILLIAC IV program. The SSK Simulator runs on the B6700.

```
SSK infilename,listfilename{,arealist}↵
```

Keyword Arguments: The following keyword arguments may be inserted in the above syntax in any position. All arguments must be separated by commas:

```
VALUE=taskvalue
```

```
MAXTIM=time
```

```
DMPFIL=dumpfilename
```

where

- infilename is the name of an ILLIAC IV SAVE (ISV) file containing the ILLIAC IV program to be simulated. If the infilename is entered without an extension, the default extension ISV will be used.
- listfilename is the user-assigned name for the output file from SSK.
- arealist is a list of up to eight entries (separated by commas), each of the form

  ```
  I4DMn=filename
  ```

  where n is an integer from 0 to 7. The filenames are the names of data files to be accessed by SSK, in place of ILLIAC IV Disk Memory transfers. Data transfer calls in the program should refer to I4DM0, I4DM1, I4DM2, etc. as I4DM areanames. (See "Data Input/Output" below.)

- taskvalue specifies the timing options selected, according to the following table:

taskvalue	Timing Option
1	Static timing only
2	Average number of PE's enabled
4	Dynamic timing without memory conflict
8	Dynamic timing with memory conflict
16	Detailed timing

  Any combination of these basic options can be specified by a taskvalue which is the sum of two or more of the entries in the above table — for example, to select static timing, average number of PE's enabled, and dynamic timing with memory conflict, the taskvalue would be 11 (i.e., 1+2+8). If VALUE=taskvalue is omitted from the statement, a taskvalue of 1 is assumed as the default value. A detailed description of the timing information returned is provided in the SSK User's Manual.
- time is the number of seconds that the simulation will be allowed to run on the B6700. If time is exceeded, the program will be terminated. If MAXTIM=time is omitted, no time limit is set.
- dumpfilename is the name of a file to contain a simulation dump at termination of the simulation (for any reason). This file has the form of an ordinary ISV file and may be input to SSK as the infile in a subsequent SSK statement. If dumpfilename is entered without an extension, the default extension ISV will be used. If DMPFIL=dumpfilename is omitted, no dump file is produced.

Description: SSK simulates ILLIAC IV execution of a user's program, and writes out in the designated listing file various performance timing statistics and information requested in DISPLAY statements in the program. Data transfers between Array Memory and ILLIAC IV Disk Memory are not time simulated, but READ calls in the ILLIAC IV program are simulated (see "Data Input/Output" below).

SSK simulates programs written in GLYPNIR or ASK source code language, compiled and/or assembled to ASK object code, and link-edited with LINKED. The program is "loaded" and "executed" exactly as if it were being run on the ILLIAC IV, except for data transfers as noted above.

If the SSK statement includes a MAXTIM=time argument, specifying a maximum number of seconds, execution of SSK on the B6700 will be terminated after the specified number of seconds. If a DMPFIL=dumpfilename is included, a simulated ILLIAC IV SAVE (ISV) file is written out to the specified file upon termination of the simulation (either under program control, or for any other reason such as the expiration of the specified maximum time). This dump has the form of an ordinary ISV file, and can be input to SSK as the infile in a subsequent SSK statement; the effect of this is to restart the simulation where it left off (assuming that it did not run to completion).

Data Input/Output:

If the source program was written for the ILLIAC IV, then data transfers within the program are specified by areaname (see MAP). SSK does not use the ILLIAC IV Disk Memory. In order to avoid modifying the ILLIAC IV program, the user, to run with SSK, should name his I4DM areas I4DM0, I4DM1, ..., I4DM7. (SSK can only simulate up to eight ILLIAC IV disk areas.) In the SSK statement a correspondence is set up, by the user, between each area and a data file. SSK will then handle a READ call that specifies an areaname (e.g., I4DM3) by reading or writing the associated data file.

Data File Conversion: The input data files may be created using the ACL CONVRT subsystem. This subsystem converts input data files written in either ASCII code or B6700 word formats into ILLIAC IV word formats.

```
CONVRT infilename,outfilename,{SIZE=nn}{,TYPE=intype}b
```

where

infilename is the name of the input data file which is to be converted into ILLIAC IV word formats.

outfilename is the name of the output data file where the ILLIAC IV formatted data is to be written.

SIZE=nn specifies the ILLIAC IV word size required for the converted output data; where

nn = 32	means ILLIAC IV 32-bit format
nn = 64	means ILLIAC IV 64-bit format

If SIZE=nn is omitted, the 64-bit word format is selected.

TYPE=intype specifies the input data format type where

intype =	CHARACTER means the input data is ASCII coded numeric data
intype =	SINGLE means the input data is B6700 single-precision binary
intype =	DOUBLE means the input data is B6700 double-precision binary.

If TYPE=intype is omitted, the input data is assumed to be ASCII coded numeric data.

The SUBMIT statement enters a request for the processing of a specified file of ACL statements in the batch queue.

```
SUBMIT pifname,{pofname},{runcode},userid,password,accountb̸
```

where

- pifname is the name of a file containing a sequence of ACL statements, to be used as a "primary input file" or PIF.
- pofname is the name of a file to contain all system messages and certain listings produced as a result of processing the PIF. This file is called the "primary output file" or POF. If pofname is omitted from the SUBMIT statement, the default pofname will have the same name as pifname, with "POF" as its extension.
- runcode is a one-digit number from 0 to 3 indicating the system resources required by the job being submitted. If the PIF contains a RUN statement or a MOVE statement, the ILLIAC IV resource is required. If it contains any GLYP, ASK, or SSK statements, the B6700 resource is required.

runcode	Resources Required
0	Neither ILLIAC IV nor B6700
1	ILLIAC IV but not B6700
2	B6700 but not ILLIAC IV
3	Both ILLIAC IV and B6700.

  If runcode is omitted, the default value is 3.
- userid is the user identification under which the PIF is to be processed.
- password is the password associated with userid.
- account is a string of digits chosen by the user.

Description:

SUBMIT is the ACL statement which enters a job request (associated with a file of ACL statements) into the ILLIAC IV System batch job queue. During the initial implementation of the ILLIAC IV System, all statements which require access to the B6700 or the ILLIAC IV resources can only be entered as part of a file submitted in this fashion. Any other statements except OUTPUT, NEWS, and NOTIFY _may_ be submitted in a batch processed ACL file.

The SUBMIT statement is normally entered interactively from a user's terminal, although it may be entered in a previously submitted batch job. Upon receiving the SUBMIT statement, the ILLIAC IV System will respond by outputting a _jobid_ on the user's terminal (or in the user's POF if the SUBMIT is part of a batch sequence). The _jobid_ is used for subsequent batch job control (see INQ, DELJOB).

The sequence of ACL statements in the file named _pifname_ is deferred for subsequent scheduling and processing by the ILLIAC IV System. It will not be processed interactively. The user may continue his ACL interactive sequence or may log out to await processing of the batch submission.

The primary input file (PIF) may be entered into the system in any of the normal ways a file may be created; e.g., it may be built at a user Host and transferred to the ILLIAC Host, or it may be interactively created at the ILLIAC Host using the DED text-editing subsystem. The PIF can contain any sequence of ACL statements (except OUTPUT, NEWS, or NOTIFY), terminated either by EOF or by a LOGOUT statement.

Example 1:

```
SUBMIT MYJOB.PIF,OUTFILE.POF,3,JONES,GILES,4920b̸
ID=89
```

where

MYFILE.PIF is the primary input file, OUTFILE.POF is the primary output file, the job requires access to both the ILLIAC IV and the B6700, JONES is the user identification, the password is GILES, and 4920 is the account. 89 is the job identification assigned by the system to this job.

The TEST statement, used in a file of ACL commands or a macro, numerically evaluates an expression and branches to one of three labeled ACL statements within the file or macro, depending on whether the value of the expression is less than, equal to, or greater than zero.

```
TEST expression,({neglabel},{zerolabel},{poslabel})b̸
```

where

- expression is

  (1) any previously defined control parameter;
  (2) the STATE parameter;
  (3) any legal expression (see Section 6.2.5).

- neglabel is the ACL statement label to branch to if the evaluation of the expression is less than zero. If omitted, and expression < 0, the branch is to the next statement following the TEST statement.
- zerolabel is the ACL statement label to branch to if the evaluation of the expression is zero. If omitted, and expression = 0, the branch is to the next statement following the TEST statement.
- poslabel is the ACL statement label to branch to if the evaluation of the expression is greater than zero. If omitted, and expression > 0, the branch is to the next statement following the TEST statement.

Description: TEST determines the value of expression and jumps to neglabel if expression < 0, to zerolabel if expression = 0, or to poslabel if expression > 0. These labels may reference forward or backward in the ACL statement sequence.

The TEST subsystem takes no action when initiated from the terminal.

The values of previously defined control parameters may be used within the expression. If an undefined control parameter is encountered by TEST, unsigned zero is used for its value and it remains undefined.

A special control parameter, called STATE, has a reserved definition in ACL. The STATE parameter is a word of information returned by the subsystem called in the user's most recent ACL statement. In particular, if the sign of STATE is negative, the statement preceding the TEST statement aborted abnormally. With the exception of EQU and TEST, every ACL statement causes a new value to be assigned to STATE.

When the TEST statement is used in a macro or in a file of ACL statements to be processed via an ALT statement, the dollar-sign character ($) may be used in place of neglabel, zerolabel, or poslabel. The branch will then be not to a labeled statement in the same file or macro, but rather to the statement immediately following the ALT statement or macro-call; in other words the $ causes a return from the file or macro. It cannot be used in a primary input file because there is no meaning to a return from a batch job; thus the $ does not have the effect of a LOGOUT statement.

Example 1:

```
TEST LIST,(RETURN,EXIT,ERROR)ƀ
```

The ACL statement labeled RETURN will be branched to if the value of the control parameter LIST is less than zero; the statement labeled EXIT will be branched to if the value of LIST is equal to zero; and the statement labeled ERROR will be branched to if the value of LIST is greater than zero.

Example 2:

```
TEST NUMBER+COUNT-2,(BACKUP,NOGOOD,EXIT)ƀ
```

Here the expression NUMBER+COUNT-2 is tested, where NUMBER and COUNT are control parameters defined in preceding EQU statements, and therefore have integer values. The expression is evaluated arithmetically; if its value is less than zero the system will branch to an ACL statement labeled BACKUP; if it is equal to zero, it will branch to a statement labeled NOGOOD; if it is greater than zero, the system will branch to the statement labeled EXIT.

Example 3:

```
TEST LASTCH(A,B,EX),(OKAY,ABORT,RETURN)ƀ
```

Here the expression is replaced by the macro-call LASTCH(A,B,EX), where LASTCH is the name of a previously defined macro and A, B, and EX are the names of control parameters defined in preceding EQU statements, now being passed as arguments to LASTCH. It is assumed that the text of the macro LASTCH is an expression which, when supplied the arguments A, B, and EX, will result in an integer value. If this value is less than zero, the system will branch to the ACL statement labeled OKAY; if it is equal to zero, the system branches to statement ABORT; and if it is greater than zero, the system branches to statement labeled RETURN.

Example 4:

```
:
TEST OLDNUM+NEWNUM,($,ZOT,NEXT)b̸
NEXT:EQU NEWNUM,OLDNUM+NEWNUMb̸
:
ZOT:ALT ZEROCASE.BAILOUTb̸
```

These statements are assumed to be part of a sequence of ACL statements in a file named DO.NOW, which is branched to by an ALT statement in another file named CALLFILE.MASTER. The TEST statement in file DO.NOW evaluates the expression OLDNUM+NEWNUM (where OLDNUM and NEWNUM are previously defined control parameters). If the value of the expression is negative, the next statement processed will be the one immediately following the ALT statement in file CALLFILE.MASTER; if it is zero, the next statement processed will be the one labeled ZOT, which branches to another file of commands named ZEROCASE.BAILOUT; if the expression has a positive value, the statement labeled NEXT will be processed, causing the value of OLDNUM+NEWNUM to be assigned to NEWNUM.

The UDIR subsystem returns information on a file or files in the user's directory that have been copied to the UNICON, i.e., "archived" (see Section 4.4.1). Since retrieval of archived files can be done much more quickly when exact information is sent to the system operator (see Section 4.4.2), the user is advised to execute UDIR routinely and save the output.

$\boxed{\text{UDIR↵}}$

Description: UDIR prompts the user by typing "OUTPUT TO:". To specify output to a file, respond by typing a <u>filename</u> terminated by a carriage return; to specify output to the terminal, respond by typing a carriage return.

UDIR next prompts by typing "FILE NAME:". To get information on a specific file, type the <u>filename</u> and terminate with a carriage return. To get information on all files in your directory that have been archived, type a carriage return.

For each file, UDIR responds by typing the <u>filename</u>, the page count, the date and time when the file was archived, and the UNICON address.

Example:

```
!UDIR↵
OUTPUT TO:↵
FILE NAME:PROG.SUBRS;2↵

        <JONES>
PROG.SUBRS;2   4 PGS   FRI 11 JUL 73   15:04 -- 00021,16571,5

!
```

(Underlining indicates material typed by the system.)

SECTION 8

ACL JOBS AND USAGES

SECTION 8

ACL JOBS AND USAGES

CONTENTS

SECTION 8

ACL JOBS AND USAGES

8.1 THE ACL JOB

A job is described by a sequence of ACL statements. ACL statements may be transmitted directly from the user's terminal to the ACL executive, causing immediate initiation of ACL subsystems, or they may be contained in files.

A file of statements is passed to the ACL executive by either of two ACL subsystems, ALT or SUBMIT. A file passed via the SUBMIT subsystem becomes the primary input source of a batch job, and is then called the primary input file or PIF. A file passed via the ALT subsystem becomes an alternate input source of either a batch job or an interactive job.

A batch job is defined as a job whose primary input source is a file (passed via SUBMIT). A batch job cannot take any input directly from the user's terminal. It may use other files (passed to the ACL executive via ALT) as alternate input sources.

An interactive job is defined as a job whose primary input source is the user's terminal. It may also use files as alternate input sources, if they are passed via ALT. The interactive job begins with the user's LOGIN statement and ends with his LOGOUT statement.

These definitions merely serve to distinguish between the two categories of jobs; there are further characteristics and attributes associated with batch and interactive jobs which are explained in this section.

8.1.1 IDENTIFICATION OF A JOB

Associated with every job is a userid entered by the user in his LOGIN statement (for an interactive job) or in the SUBMIT statement (for a batch job). This userid corresponds to an entry in a permanent table maintained by the system, and is associated with a particular user. However, an individual's userid may be shared by several users; if these users are logged in at the same time, several different jobs with the same userid will be running concurrently.

The userid associated with a job is used by the system to identify the file space assigned to the job and in certain cases to assign privileges. It is also used for accounting purposes by the system. Thus concurrent jobs with the same userid will share the same file space, i.e., they will have the same file directory and will have access to the same set of files at all times (on a first-come, first-served basis in case of conflicts).

Also associated with every job is an account, which is not used by the system in the initial implementation. The account is intended for use as an accounting subcategory under the userid.

The unique identification of each job is its jobid. This is an identifier generated by the system at the initiation of every job; for example, if there are two or more concurrent jobs with the same userid, they will have different jobid's. The jobid of an interactive job is transparent to the user but is associated with everything he does during his interactive session.

In one sense a job is defined within the system as everything associated with a unique jobid, including the userid (not necessarily unique) that identifies a file space and a single account (not necessarily unique). When the interactive or batch job terminates, the jobid is purged, along with all information associated only with the jobid. (But the userid is permanent, therefore the file space and the files it contains are also permanent.)

The information associated uniquely with the jobid includes the following:

- Definitions and values of all control parameters defined by EQU statements in the same job.
- The definition and value of a STATE parameter.
- Definitions of all macros defined by MACRO statements in the same job.
- Assorted tables and other information associated with the job but not referenced explicitly by the user.

When an ALT statement is included in a job to process ACL statements contained in a file, the processing of these statements is part of the same job that contains the ALT statement. Moreover, the file of statements may contain another ALT statement to another file, and so forth to arbitrary level; everything processed as a result of any ALT statement is part of the same job in which the ALT statement occurs. The same is true for macro-calls; a macro-call may invoke a macro that consists of a sequence of ACL statements, and the processing of these statements will be part of the same job.

8.1.2 CONSTRUCTING A JOB

A batch job and an interactive job are in most respects exactly alike. This section is concerned with listing and explaining the differences.

The SUBMIT statement references a file of ACL statements and places in the batch job queue a request for processing of this file. The SUBMIT statement may occur in another batch job or in an interactive job, and specifies the following:

- The name of the file to be used as the PIF for this batch job.
- The userid to be associated with the batch job. This may be the userid associated with the job containing the SUBMIT statement that creates the batch job, and in this discussion we will assume that this is the case. Accordingly the batch job will use the same file space and file directory as the job that created it.
- An account to be associated with the batch job. This has exactly the same meaning as the account supplied for an interactive job and may in fact be the same number.
- Other information not relevant to this discussion.

Once initiated with a SUBMIT statement, the batch job is independent of the job that created it. It is a separate job. If the SUBMIT statement is entered by the user in an interactive job, the user may log out, terminating his interactive job, and this will in no way interfere with the batch job. The user may obtain information about the status of the batch job by means of an INQ statement in his interactive job (or in another batch job), and he can interfere with the batch job in two ways: he can abort it completely with a DELJOB statement, or he can alter or delete files in the shared file space that are referenced by the batch job.

Alteration or deletion of files referenced by a given job will interfere with the job. It is worth pointing out that this can happen quite easily if two or more batch jobs with the same _userid_ exist concurrently, or if two or more users are logged in simultaneously in interactive jobs with the same _userid_.

In an interactive job, system messages and responses and certain types of listings are output directly to the user's terminal. In a batch job, all such material is written sequentially into a file called the _primary output file_ or POF. This is a file in the file space associated with the batch job's _userid_; its name may be specified by the user in the SUBMIT statement or it may have a default _filename_ with the _name_ of the job's primary input file (PIF) as the _name_ and POF as the _extension_.

A batch job may contain statements that are illegal in an interactive job — i.e., statements that require access to the ILLIAC IV or B6700 resources.

It is important to bear in mind that although an interactive job is delimited by the user's LOGIN and LOGOUT from his terminal, and its primary input source is the stream of ACL statements entered from the terminal, it may also contain any number of ALT statements which initiate the processing of files of ACL statements under the same _jobid_. Likewise, the primary ACL sequence of a batch job is the PIF, but the job is not limited to the PIF; exactly like an interactive job, it may also include ALT statements referencing files of ACL other than the PIF.

Statements requiring access to ILLIAC IV or B6700 resources may not appear anywhere in an interactive job. This restriction applies equally to a file of ACL initiated by an ALT statement within an interactive job.

Batch jobs are typically more complex in structure than interactive jobs, simply because they do access ILLIAC IV and B6700 resources and because they must function properly without the user's intervention.

More complex structures can be built into a file of ACL statements than can be entered directly to the ACL executive, using labeled statements and the EQU and TEST statements. These structures are normally associated with batch jobs. However, even though the primary input source of an interactive job is not a file and therefore cannot meaningfully contain labeled statements, an interactive job can still make use of conditional loops, tests, etc. in files initiated via the ALT statement.

8.2 EXAMPLES

These examples are included for the purpose of demonstrating how a primary input file might be constructed, and the ways in which various ACL statements might be used.

8.2.1 EXAMPLE 1: SUBMITTING A BATCH JOB

The following example illustrates an ACL sequence used to submit a simple batch job for processing. User Jones at the Moon Host wishes to have a file containing GLYPNIR source code compiled, and transfer the resulting listing back to his directory at the Moon Host. Note that the primary input file could be constructed directly at the ILLIAC System using DED, in which case the first CPYNET statement below would not be necessary.

```
@LOGIN JONES  123ƀ
```

User Jones logs in.

```
JOB 74 15-JAN-73 10:30ƀ
```

The System provides the user a jobid, 74.

```
!CPYNET (MYFILE,MOON,JONES,BANJO,123),COMP.PIFƀ
```

The primary input file MYFILE has been created at the Moon Host and is now copied from Jones' directory at the Moon Host to a file named COMP.PIF in Jones' directory in the central file system. (BANJO is Jones' password.)

```
!SUBMIT COMP.PIF,,2,JONES,BANJO,123ƀ
ID=89ƀ
```

The PIF is submitted for batch processing. The primary output file will have the default name COMP.POF. The jobid assigned by the System is 89.

!LOGOUTƀ

The job is in the batch queue and requires no further intervention, so the user logs out.

The primary input file, COMP.PIF, contains the following sequence of ACL statements:

CPYNET (SOURCE,MOON,JONES,BANJO,123),ALPH.GLYPNIRƀ

Copy the file of GLYPNIR source code named SOURCE from Jones' directory at the Moon Host to a file named ALPH.GLYPNIR in Jones' directory in the central file system.

GLYP ALPH.GLYPNIR,OUT.OBJ,LIST.GLYPNIR,ASSEMBLE=NOƀ

Compile the GLYPNIR source code in file ALPH.GLYPNIR, output the ASK source code to file OUT.OBJ, and the listing to file LIST.GLYPNIR. Do not call the assembler. Syntax and error messages will be placed in the user's primary output file COMP.POF.

CPYNET LIST.GLYPNIR,(TEST,MOON,JONES,BANJO,123)ƀ

Copy the listing to file TEST in Jones' directory at the Moon Host.

CPYNET COMP.POF,(ERRORS,MOON,JONES,BANJO,123)ƀ

Copy the primary output file to file ERRORS in Jones' directory at the Moon Host.

8.2.2 EXAMPLE 2: PRIMARY INPUT FILE

The following example illustrates a sequence of ACL statements in a primary input file. Figure 8-1 (at the end of Section 8) is a flowchart for the sequence. The user wishes to compile, assemble, and execute a GLYPNIR source program. It is assumed that the PIF has been created at the user's Host and then transferred to the central file system by means of a CPYNET statement used interactively (see Example 1).

Note: The specific use of the STATE parameter in the following example is unrealistic but is included so as to demonstrate ACL's test-and-loop capability. For details see Section 8.2.2.1 below.

EQU RETRY,2ƀ

Initialize the retry counter. (An operation that fails is retried once.)

CPYNET (PROGRAM,MOON,JONES,BANJO,123),PROG.GLYPNIRƀ

The file containing GLYPNIR source code (PROGRAM) has been created at the Moon Host, and is now copied from Jones' directory there to a file named PROG.GLYPNIR in the central file system.

CPYNET (INFILE,MOON,JONES,BANJO,123),DATA.INPUTƀ

Copy the file containing input data (INFILE) from Jones' directory at the Moon Host to file DATA.INPUT in the central file system.

COMP:GLYP PROG.GLYPNIR,PROG.OBJ,PROG.LISTƀ

Compile and assemble the GLYPNIR source code; write the ASK object code in file PROG.OBJ and the listing in file PROG.LIST.

TEST STATE,(CHECK,NEXT,NEXT)ƀ

Test the STATE parameter to see if the previous subsystem (GLYP) aborted catastrophically. If so (STATE is negative), go to CHECK to see if it has been retried. If STATE is zero or positive, go to the statement labeled NEXT.

CHECK:EQU RETRY,RETRY-1ƀ

Decrement the retry counter.

TEST RETRY,(,DO2,COMP)ƀ

If RETRY=0, the operation has been retried, in which case go to an ALT statement labeled DO2; if RETRY is positive, go to COMP to try the GLYPNIR compilation again.

NEXT:EQU RETRY,2ƀ

Initialize the retry counter.

LINKED ,EXECUTEƀ

Convert the relocatable ASK object code into an ISV file named EXECUTE.ISV. Since no file is specified for LINKED control statements, these statements (i.e., INCLDE and END) follow in-line.

INCLDE PROG.OBJƀ

Take the relocatable ASK object code from file PROG.OBJ.

ENDƀ

Terminate the sequence of control statements to the LINKED subsystem.

ASSIGN:ALLOC DISK.MAPƀ

Assign I4DM space as specified in DISK.MAP. DISK.MAP is a file containing the user's I4DM layout description created at some earlier time by the MAP subsystem.

TEST STATE,(CHECK2,NEXT2,NEXT2)b̸

Test the STATE parameter to see if the previous subsystem (ALLOC) aborted catastrophically. If so (STATE is negative), go to CHECK2 to see if it has been retried. If STATE is zero or positive, go to the next statement (labeled NEXT2).

CHECK2:EQU RETRY,RETRY-1b̸

Decrement the retry counter.

TEST RETRY,(,DO2,ASSIGN)b̸

If RETRY=0, the operation has been retried, in which case go to an ALT statement labeled DO2; if RETRY is positive, go to ASSIGN to try the ALLOC operation again.

NEXT2:MOVE DATA.INPUT,I4DM:INAREAb̸

Copy the input data into a disk area (INAREA) declared in a previous MAP subsystem call.

RUN EXECUTEb̸

Load the ISV file EXECUTE.ISV and initiate its execution.

MOVE I4DM:OUTARA,DATA.OUTPUTb̸

Copy the ILLIAC IV process output data from OUTARA (a disk area declared in a MAP subsystem call) to a file called DATA.OUTPUT in the central file system. The user can now copy this output data, GLYP listing file, and primary output file back to his Host.

LOGOUTb̸

Terminate the job.

DO2:ALT CLEANUP.ACLb̸

Initiate processing of the ACL statements contained in the file named CLEANUP.ACL.

8.2.2.1 Note on the STATE Parameter

As indicated above, a negative value of STATE indicates a "catastrophic failure" of the previous subsystem. More specifically, it indicates a termination not specified by the subsystem itself, such as an interrupt from a higher level in the system. It is presently planned that ACL subsystems will use various bits of STATE in a consistent manner to indicate specifically whether the subsystem terminated normally, aborted for a specific reason, etc.; however, this has not yet been implemented. When implemented, this consistent usage of STATE will permit the user to make more useful tests than are shown in the above example (e.g., failure of ALLOC because of insufficient I4DM space).

8.2.3 USE OF MACRO IN A PIF

The following example illustrates the use of a macro in a primary input file. The sequence of statements used in the previous example to test for catastrophic failure of a subsystem can be replaced by one macro-call. The macro is defined as follows:

```
MACRO TLOOP,(LABEL)ƀ
[TEST STATE,(FAIL,OK,OK)ƀ
FAIL:EQU RETRY,RETRY-1ƀ
TEST RETRY,(,DO2,LABEL)ƀ
OK:EQU RETRY,2]ƀ
```

where

LABEL is the label of the statement which will be retried once if a catastrophic failure occurs.

Using the previous example, the macro-calls would be inserted as follows:

```
EQU RETRY,2ƀ
CPYNET (PROGRAM,MOON,JONES,BANJO,123),PROG.GLYPNIRƀ
CPYNET (INFILE,MOON,JONES,BANJO,123),DATA.INPUTƀ
COMP:GLYP PROG.GLYPNIR,PROG.OBJ,PROG.LISTƀ
TLOOP(COMP)ƀ
LINKED ,EXECUTEƀ
INCLDE PROG.OBJƀ
ENDƀ
ASSIGN:ALLOC DISK.MAPƀ
TLOOP(ASSIGN)ƀ
MOVE DATA.INPUT,I4DM:INAREAƀ
RUN EXECUTEƀ
MOVE I4DM:OUTARA,DATA.OUTPUTƀ
LOGOUTƀ
DO2:ALT CLEANUP.ACLƀ
```

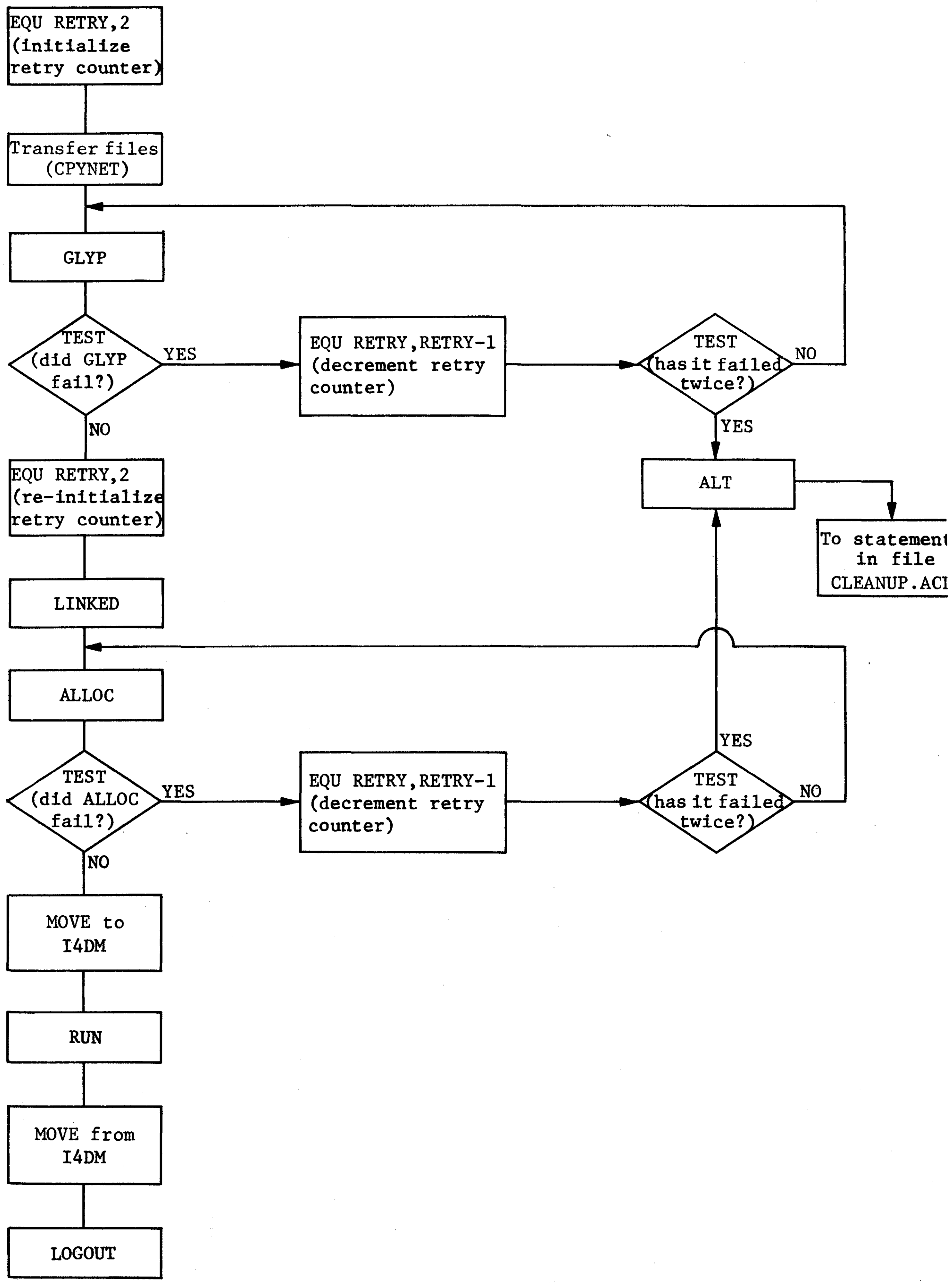

EQU RETRY,2 (initialize retry counter)
Transfer files (CPYNET)
GLYP
TEST (did GLYP fail?)
YES
NO
EQU RETRY,RETRY-1 (decrement retry counter)
TEST (has it failed twice?)
NO
YES
ALT
To statement in file CLEANUP.ACI
EQU RETRY,2 (re-initialize retry counter)
LINKED
ALLOC
TEST (did ALLOC fail?)
YES
NO
EQU RETRY,RETRY-1 (decrement retry counter)
TEST (has it failed twice?)
YES
NO
MOVE to I4DM
RUN
MOVE from I4DM
LOGOUT

Figure 8-1. Flowchart for Ex

APPENDIX A

DED TEXT EDITOR

APPENDIX A

DED TEXT EDITOR

This appendix to the System Guide for the ILLIAC IV User is a detailed description of the operation and use of the DED text editor.

Section A-1 describes the basic concepts and features of DED; Sections A-2 and A-3 provide detailed descriptions of the DED commands; Section A-4 describes the use of control characters in DED; and Section A-5 is a quick reference summary of DED commands.

APPENDIX A

DED TEXT EDITOR

CONTENTS

SECTION A.1

INTRODUCTION

A.1.1 GENERAL DESCRIPTION

DED is a line-oriented text entry and editing subsystem operating under ACL; it is initiated by the ACL statement DED. DED is normally used interactively from the user's terminal, and may be used to type out an existing file of text for examination, alter an existing file of text by means of various editing operations, or create a new file containing text input from the user's terminal (or a combination of these functions).

"Text" in this application means ASCII-coded character sequences. There are only two constraints on the text handled by DED:

(1) It must be ASCII coded (and may use the entire ASCII character set).

(2) It must be organized into consecutive lines separated by carriage returns.

When DED creates a file, the file contains a body of text divided into one or more lines. No other information or formatting is added by DED. Conversely, when DED reads an existing file, it is assumed to contain text only.

DED is similar to other line-oriented interactive text editors employing ASCII code, and in many cases ASCII text files created with other text editors are fully compatible with DED.

Typical applications of DED include (but are not limited to) the following:

(1) Typing out symbolic listing files.

(2) Creating and editing symbolic code files for input to compilers, assemblers, etc. In particular, DED may be used to create and edit files of ACL statements.

(3) Creating and editing files of natural-language text, such as messages, documents, etc.

A.1.2 WORKING STORAGE

Although one of the primary uses of DED is to edit the contents of an existing file, it is important to bear in mind that the editing commands of DED do not operate directly upon the file.

Instead, DED makes a temporary copy of the file and operates upon this copy. The copy is stored in an area of memory called "working storage," which is maintained by DED and is not part of the user's assigned file-storage space. Upon explicit command from the user, DED copies the total contents of working storage into a file in the user's directory (either a new file, or an old or new version of an existing file). This is the only way in which DED modifies an actual file.

Working storage can be loaded by copying a file into it as described above, or by entering lines of text into working storage directly from the keyboard. At any time when there are two or more lines of text in working storage, the editing commands can be used to modify them.

The text contained in working storage is organized into lines. One line of stored text normally corresponds to one typed line on the paper of the user's terminal. Each line position in working storage is numbered for reference (see below); working storage has a maximum capacity of 30,000 lines or 700,000 characters. If either of these limits is exceeded, some of the text will be lost.

A.1.3 THE TWO DED OPERATING MODES

DED operates in two distinct modes, called "normal mode" and "character editing mode." There is a separate set of commands for each mode.

Normal mode commands are used for reading and writing files, typing out lines of text from working storage, searching the text in working storage for lines containing a specified string, editing the text in working storage, and various miscellaneous functions. Specifically, the editing functions performed in normal mode are the following:

- Enter lines of text into working storage, either from a file or from the keyboard.
- Delete lines of text from working storage.
- Copy lines of text from one location to another within working storage.

- Edit the text within a line by making a string substitution, i.e., searching for a specified string of characters and replacing it with another specified string.

The string substitution method is a powerful means for editing the text of a line, but does not permit the user to edit on a character-by-character basis. Character-by-character editing is done only in the "character editing mode." A normal mode command ("C") causes DED to go into character editing mode to edit a specified line in working storage; the user may then enter a series of character editing commands to modify the text in the line, returning to normal mode by means of an explicit command.

The normal mode commands are fully described in Section A.2, and the character editing mode commands in Section A.3.

A.1.4 LINES

As noted above, the text in working storage is divided into lines. It is necessary to distinguish clearly between a line in working storage and a typed line (either input or output) on the user's terminal, as they are not the same and do not necessarily correspond.

A line in working storage is a sequence of ASCII characters, of any length; specifically, it is not limited to the line width of the user's terminal. When a line is longer than the line width on the user's terminal, DED will automatically break it up into two or more lines at the terminal, but it remains a single line in working storage (see below).

Lines in working storage are separated by user-supplied carriage returns, and their positions are consecutively numbered as explained below in Section A.1.5. The term "line," as used in this guide, always refers to a line in working storage, not to a typed line at the terminal, unless explicitly referred to as a "terminal line," meaning a single line typed on the terminal.

When DED types out a line in working storage that is too long to fit on a single terminal line, it automatically breaks the line into two or more terminal lines. Each break is made by inserting into the typeout a carriage return followed by two asterisks (**), which appear at the beginning of the next terminal line. The two asterisks at the beginning of a terminal line are an indication that the line is a continuation of the preceding terminal line, i.e., part of the same line in working storage. The carriage return and asterisks are only inserted into the typeout — the line in working storage is not altered in any way.

The same thing is done when the user is typing in a line and it becomes too long to fit on a single terminal line: DED types out a carriage return followed by two asterisks, and the user may continue typing. This may go on through any number of terminal lines until the user types a carriage return, which ends the line. All of the terminal lines are then taken by DED to be a single line, which is placed in working storage.

A.1.5 LINE NUMBERS AND THE LINE POINTER

As noted above, the positions occupied by lines in working storage are consecutively numbered. The first line is always Line 1.

The number of a line position is referred to as a "line number." It is important to understand that a line number is not a permanent identifier for a particular line of text, since editing may cause a line to be moved to a new position. The line number identifies the position of a line in working storage, not its content.

In this document, the term "line number" always refers to a position; the term "line" refers to a particular line of text. In entering a DED command in normal mode, the user identifies a line by reference to its line number, i.e., its present position relative to the other lines in working storage. (There is also a command that allows the user to locate a particular line by reference to its content, i.e., search for a line starting with a specified character string.)

DED maintains a line pointer that always points to one of the line positions in working storage. The line pointed to by the line pointer is called the "current line." The line pointer may be moved explicitly by the user, and it is moved by DED as part of the processing of certain commands. In the command descriptions in Section A.2, the position of the line pointer after the processing of each command is explicitly noted where applicable.

Many normal mode commands take a line number as an optional portion of the command, to be supplied by the user. This is indicated by the notation {#} in the syntax descriptions in Section A.2. The line number in these commands specifies the line to be operated on by the command, or the starting line for a search. In all of these commands, the user may omit the line number, in which case the current line is used by default. In character editing mode, the line pointer remains at the line specified by the user in entering character editing mode.

A.1.5.1 SPECIFYING LINE NUMBERS

In the command descriptions in Section A.2, the symbol {#} is used to indicate that the user may optionally specify a line number. The effect of this specification is to update the line pointer; the command then operates upon the new current line, i.e., the line in the position specified by the user. If the specification is omitted, the line pointer is unchanged and the command operates upon the current line.

If the specification is a single decimal integer, this number becomes the new current line number. For example, the character "/" is a command that causes the current line to be typed out on the user's terminal. The command 52/ causes the line pointer to be set to Line 52 and Line 52 typed out.

If the specification is a plus or minus sign followed by an integer, the line pointer is incremented or decremented by the specified amount. For example, if the current line is 15, the command +1/ causes the line pointer to be set to 16 and Line 16 typed out. Then, if the next command is -2/, it will cause the line pointer to be set to 14 and Line 14 typed out.

The largest possible line number is the number of the last line in working storage. If the user specifies a larger number, the line pointer is set to the last line in working storage.

A.1.6 DED PROMPT CHARACTERS

In normal mode, DED uses two prompt characters with distinct meanings. A colon (:) prompt indicates that DED is ready for a normal mode command, while an asterisk (*) prompt indicates that DED is ready for the user to type in a sequence of text characters. This sequence may be a line of text to be placed in working storage, or it may be a string to be searched for or substituted, depending upon the command being executed. DED does not prompt in character editing mode.

A.1.7 <u>BASIC OPERATING PROCEDURES</u>

The following is a basic outline of the procedures for using DED.

(1) Enter the DED statement from ACL. This initiates the DED subsystem, which types out a herald and then a colon prompt, indicating that it is ready to receive a DED command.

(2) You may now either read a file into working storage or begin entering lines of text from the terminal, as follows:

(a) To read a file into DED working storage, type the letter R (the DED "READ" command). DED will respond by typing

```
INPUT FILE:
```

Type in the <u>filename</u> (including the <u>extension</u>) of the file to be read, terminated with a carriage return. DED will respond with

```
[OLD VERSION]
```

Type a second carriage return to confirm. DED will first copy the contents of the file into working storage, and then type out the number of lines in the file, followed by a colon prompt to show that it is ready for the next command. The commands described in Sections A.2.2 and A.3 may now be used to edit the text in working storage.

(b) To enter lines of text from your terminal into working storage, type the letter A (the DED "APPEND" command). DED will respond with an asterisk prompt to show that it is ready to receive text. Any number of lines of text can now be typed in. Each line is terminated with a carriage return, except the last line, which is terminated with a ↑Z ("control" and "Z" keys depressed together — see Section A.4). DED will then respond with a colon prompt to show that it is ready for another command. The text lines are now in working storage.

(3) When two or more lines of text are in working storage, you may use the editing commands described in Sections A.2.2 and A.3.

(4) To write the contents of working storage into a file, type the letter W (the DED "WRITE" command). DED will respond with

    OUTPUT FILE:

Type in the <u>filename</u> (including the <u>extension</u>) of the output file, terminated with a carriage return. DED will respond with:

[NEW FILE]	if the <u>filename</u> does not already exist in the user's directory
or	
[NEW VERSION]	if the <u>filename</u> already exists and the user does not supply an old <u>version</u> number as part of the <u>file-name</u>
or	
[OLD VERSION]	if the user supplies an existing <u>filename</u> with an existing <u>version</u> number.

Type a second carriage return to confirm. DED will write out the file, type out the number of lines written, and type a colon prompt and await a new command.

(5) To write the contents of working storage into a file and terminate DED, type the letter Q (the DED "QUIT" command). The action is exactly the same as the WRITE command (see above), except that DED automatically terminates after writing out the file.

(6) To terminate DED without writing a file, type the letter E (the DED EXIT command). DED will respond with

    [CONFIRM]

Type a carriage return to confirm, and DED will terminate.

A.1.8 <u>EDITING TEXT WITHIN LINES</u>

The "C" command (SET CHARACTER EDITING MODE) causes DED to go into character editing mode. In character editing mode, DED takes a different set of commands, which are described fully in Section A.3. When editing of the line is completed, character editing mode is terminated and DED returns to normal mode.

Note that the character editing mode is not the only means provided for editing text within lines. There are also two string-substitution commands (see Sections A.2.2.7 and A.2.2.8), which cause a specified text string to be searched for and replaced with a second string. These commands may be used on a single line or on a set of lines, and afford considerable power to the user.

A.1.9 CONVENTIONS AND TERMINOLOGY

The following conventions are used throughout this appendix:

- Uppercase letters, digits, and special characters must appear exactly as shown in the command descriptions.
- Information in lowercase letters is variable data to be entered by the user.
- Braces ({}) indicate that the item enclosed is optional.
- Underlined text in the examples indicates information typed out by DED, as distinguished from user-typed input. Explanatory notes appear in parentheses.

The special symbols used in this document are defined below:

Symbol	Meaning
{#}	Indicates that a line number may be specified by the user. This is always optional.
↑a	where "a" is any character — "CONTROL a"
↲ or CR	Carriage Return
LF	Line Feed

The following terms are defined according to their specialized use in this appendix.

character: Any member of the ASCII character set.

current line: The line of text in working storage that the line pointer points to at any given time.

line: A sequence of characters in working storage, terminated by a user-supplied carriage return. See Section A.1.4.

line pointer: A pointer to one of the lines into which the text in working storage is divided. This line is called the "current line."

terminal line: A single physical line typed on the user's terminal.

text: Data in the form of a sequence of ASCII-coded characters. Text in working storage is organized into lines.

working storage: The space used by DED to store a body of text. Working storage is not part of the user's assigned file storage space. Working storage may be loaded from a file or from the user's terminal. The contents of working storage may be altered by means of DED commands and may be written out to a specified file; this is the only way in which DED modifies a file. See Section A.1.3.

SECTION A.2

NORMAL MODE: COMMAND DESCRIPTIONS

DED's commands consist of one or more characters — letters, digits, or special symbols. Some commands require a terminating carriage return, indicated in the following descriptions by the symbol ↵. Others execute immediately. Do not use a terminating carriage return where it is not required, as this may have unfortunate effects.

In the following command syntax descriptions, the notation {#} is used to indicate the optional specification of a line number. This specification is <u>always</u> optional; if it is omitted, the "specified line" referred to in the command description will be the current line.

For certain commands, the optional line number specification is omitted from the syntax description. However, it is <u>always permissible</u> to use a line number specification as the first element of a normal mode command. The effect of this is always the same, namely to move the line pointer to the specified line before carrying out the command. The specification is omitted from the syntax description for commands where there is ordinarily no reason to specify a line number.

All normal-mode DED commands, divided into functional categories, are described in this section. Each category has the following format:

(1) A general description of the functions performed by this set of commands.

(2) A listing of each command in this category, describing the syntax and function of each.

A.2.1 <u>TYPEOUT COMMANDS</u>

The typeout commands cause the contents of one or more lines or the current line number to be printed on the terminal. All commands that cause lines to be typed out also cause the line numbers to be typed at the beginning of each output line, with the exception of the P command. Note that the line number is not part of the stored content of the line. The typing out of line numbers can be suppressed by means of an 04 command (see Section A.2.5.3).

A.2.1.1 TYPE LINE NUMBER OF CURRENT LINE

Syntax: =

Description: Type out the number of the current line. (Line 1 is the first line of text in working storage.)

Example:

```
:=17
:
```

A.2.1.2 TYPE SPECIFIED LINE

Syntax: {#}/

Description: Position the line pointer to the specified line and type the line.

Example: Print 9th line of text, then increment line pointer and print 17th line.

```
:9/
9 NINTH LINE OF TEXT
:+8/
17 SEVENTEENTH LINE OF TEXT
:
```

A.2.1.3 TYPE PRECEDING LINE

Syntax: ↑

Description: Position the line pointer to the line preceding the current line and type the line.

A.2.1.4 TYPE FOLLOWING LINE

Syntax: LF (line feed)

Description: Position the line pointer to the line following the current line and type the line.

A.2.1.5 TYPE LINE NUMBER OF LAST LINE OF TEXT AND MOVE LINE POINTER

Syntax: $

Description: Type the number of the last line in the text and move the line pointer to this line.

A.2.1.6 TYPE n LINES

Syntax: {#}Tnƀ

Description: Type out n consecutive lines beginning with the specified line. The line pointer is positioned to the last line typed out. This command can be terminated during output by typing ↑O. The line pointer is positioned to the last line typed.

Example: Type four consecutive lines beginning with Line 11.

```
:11T4
11 A QUICK BROWN FOX
12 JUMPS OVER THE LAZY DOG
13 NOW IS THE TIME
14 FOR ALL GOOD MEN TO
:=14
:
```

A.2.1.7 TYPE ENTIRE CONTENTS OF WORKING STORAGE

Syntax: P

Description: Print (type out) the entire text in working storage, omitting line numbers. The line pointer is unchanged. This command can be terminated by typing ↑O.

Example:

```
:P
THE TYPEOUT COMMANDS (first line of text)
CAUSE THE CONTENTS OF
ONE OR MORE LINES
TO BE PRINTED ON THE TERMINAL (last line of text)
4 LINES PRINTED
:
```

A.2.2 EDITING COMMANDS

The editing commands enable the user to delete one or more lines, insert text before a specified line, add text after the last line, copy lines from one position to another in the working storage area, or modify the text of one or more existing lines.

All of the lines in working storage are always numbered consecutively starting with Line 1, according to the positions they occupy. There are no gaps in the numbering sequence; if a line is deleted, all higher-numbered lines are "moved up" to fill the gap. Conversely, if a new line is inserted into the sequence, all higher-numbered lines are "moved down" to make room for it. For this reason, it is important to remember that the line number associated with any particular line may change several times during an editing session. A line number identifies only a position in working storage, not the content of the line presently occupying that position.

In the present implementation, no editing commands except "A" can be used unless there are at least two lines of text in working storage. The "A" (APPEND) command is used to insert lines of text from the keyboard into working storage.

The "R" (READ) command is not described in this section; it is described as a "file command" in Section A.2.4. However, it may also be thought of as an editing command since it modifies the text in working storage.

A.2.2.1 DELETE SPECIFIED LINE

Syntax: {#}D

Description: Delete the specified line. The line pointer is left at the line following the deleted line. If the last line is deleted, the line pointer is positioned to the new last line.

Example: Delete the last line in the above example, and type out the new text.

```
:4D
:1T4b
1 THE TYPEOUT COMMANDS
2 CAUSE THE CONTENTS OF
3 ONE OR MORE LINES
:
```

A.2.2.2 DELETE n LINES

Syntax: {#}Knƀ

Description: Delete n consecutive lines beginning with the specified line. The line pointer is left at the line following the deleted lines. If the last line is deleted, the line pointer is positioned to the new last line.

Example: Delete two consecutive lines starting at Line 3.

```
:3K2ƀ
```

Lines 3 and 4 are deleted.

A.2.2.3 INSERT TEXT BEFORE SPECIFIED LINE

Syntax: {#}I

Description: Insert one or more lines of text before the specified line. Each line except the last is terminated with a carriage return; the last line is terminated with a ↑Z, which terminates the command. The line pointer remains at the specified line regardless of the number of new lines inserted. (Note that line numbers will change after the insertion.)

See Section A.2.7 for the use of control characters while typing in text.

Example: Type out Lines 1 through 4, then insert one line before Line 4.

```
:1T4ƀ
1 THE TYPEOUT COMMANDS
2 CAUSE THE CONTENTS OF
3 ONE OR MORE LINES
4 TO BE PRINTED ON THE TERMINAL
:I
*OR THE CURRENT LINE NUMBER↑Z
:1T5ƀ
1 THE TYPEOUT COMMANDS
2 CAUSE THE CONTENTS OF
3 ONE OR MORE LINES
4 OR THE CURRENT LINE NUMBER
5 TO BE PRINTED ON THE TERMINAL
:
```

A.2.2.4 APPEND TEXT AFTER LAST LINE

Syntax: A

Description: Enter one or more lines of text (terminated by ↑Z) into working storage. If working storage is empty, the first line appended becomes Line 1; if it is not empty, the new lines are appended after the last line in working storage. Each new line except the last is terminated with a carriage return; the last new line is terminated with ↑Z, which terminates the command. The line pointer is positioned to the last new text line.

See Section A.2.7 for the use of control characters while typing in text.

Example: Enter the following four lines of text.

```
:A
*TO INPUT A BODY OF TEXT FOR THE↵
*FIRST TIME, TYPE THE LETTER A.↵
*DED PROMPTS WITH AN ASTERISK↵
*AND LINES OF TEXT CAN BE TYPED IN.↑Z
:
```

A.2.2.5 COPY n LINES

Syntax: {#}M{n}.{m}↵

where a carriage return may be used in place of the period.

Description: Copy n lines, starting with the specified line, and insert them in order above line m. The line pointer remains at the specified line. If n is omitted, every line from the specified line to the last line in working storage is copied; if m is omitted, the copied lines are inserted above the last line.

Example: Type out Lines 1 through 5, then move Lines 1 through 3 above Line 5.

```
:1T5ƀ
1 THE TYPEOUT COMMANDS
2 CAUSE THE CONTENTS OF
3 ONE OR MORE LINES
4 OR THE CURRENT LINE NUMBER
5 TO BE PRINTED ON THE TERMINAL
:1M3.5ƀ
:T8ƀ
1 THE TYPEOUT COMMANDS
2 CAUSE THE CONTENTS OF
3 ONE OR MORE LINES
4 OR THE CURRENT LINE NUMBER
5 THE TYPEOUT COMMANDS
6 CAUSE THE CONTENTS OF
7 ONE OR MORE LINES
8 TO BE PRINTED ON THE TERMINAL
:
```

A.2.2.6 SET CHARACTER EDITING MODE

Syntax: {#}C

Description: DED types out the specified line and goes into character editing mode, permitting the user to edit the text of the specified line on a character-by-character basis. In character editing mode, DED accepts the character editing commands described in Section A.3. When the editing of the line is completed, one of several character editing commands may be used to terminate character editing mode and return to normal mode. At termination of character editing mode, the line pointer remains at the specified line.

Note: There are three ways to edit the text within a line: the character editing commands and the two SUBSTITUTE commands, "S" and "Z" (described below). In many cases, a SUBSTITUTE command is more convenient and efficient than the character editing commands.

A.2.2.7 SUBSTITUTE A STRING IN SPECIFIED LINE

Syntax: {#}S{string1}↑Zstring2ƻ

where each string may be terminated by either ↑Z or ƻ.

Description: Find the first occurrence of string2 in the specified line, and replace it with string1; the line pointer is left at the specified line. If string2 is found, DED types out 1 SUBSTITUTION MADE; if string2 is not found, DED types out SEARCH FAILURE. If string1 is omitted, string2 is erased.

Example: Substitute "terminal" for the first occurrence of "teletype" in the current line.

```
:/ (print current line)
8 TO BE PRINTED ON THE TELETYPE
:S*TERMINAL↑Z*TELETYPEƻ
1 SUBSTITUTION MADE
:/ (print current changed line)
8 TO BE PRINTED ON THE TERMINAL
:
```

A.2.2.8 SUBSTITUTE A STRING IN n LINES

Syntax: {#}Z{n}.{string1}↑Zstring2ƻ

where a carriage return may be used in place of the period and/or ↑Z.

Description: Replace every occurrence of string2 with string1 in the next n lines (starting with the specified line). The line pointer is positioned to the last line searched. If n is omitted, every occurrence of string2 is replaced from the specified line to the last line in working storage and the line pointer is left at the last line. DED types out XX SUBSTITUTIONS MADE, where XX is the number of strings replaced. If string1 is omitted, every string2 is erased.

Example: Replace every occurrence of "THE" with "YOUR" in Lines 1 through 5.

```
:1Z5.*YOUR↑Z*THE↲
4 SUBSTITUTIONS MADE
:
```

A.2.3 SEARCH COMMANDS

The search commands cause DED to position the line pointer to a line starting with a specified character string or to type out lines starting with or containing a specified character string.

Note that DED takes a string to be just an arbitrary sequence of characters, not necessarily a word. In formulating a search command to search for a word, remember that the word may appear as a string embedded in some other word; for example, a search for the word "train" may find the word "training" or "untrained." In some cases, this problem can be avoided by including a space at the beginning or end (or both) of the string to be searched for, as in the examples below.

A.2.3.1 JUMP TO LINE STARTING WITH SPECIFIED STRING

Syntax: {#}Jstring↲

Description: Jump to the next line in the text (starting at the specified line) that starts with the specified string of characters; the line starting with string is typed out, and the line pointer is positioned to this line. Every line in the user's text will be searched until string is found; i.e., the search will continue to the last line in working storage, then resume at Line 1 and continue to the line preceding the specified line. If string is not found, the specified line is typed out and the line pointer is left at the specified line.

Example: Find the next line beginning with the string "CAUSE " (note the space at the end of the string).

```
:J*CAUSE ↲
2 CAUSE THE CONTENTS OF
:
```

If the space were omitted, DED might instead find a line starting with the word "CAUSED"

A.2.3.2 TYPE LINES STARTING WITH SPECIFIED STRING

Syntax: {#}Lstringƀ

Description: Locate and type out lines in the text that start with the specified string of characters; the search begins at the specified line. If string is not found, the specified line is typed out. In either case, the line pointer remains at the specified line. This command can be terminated during typeout by typing ↑O.

Example: Locate and type out lines in the text beginning with the string "TO " (note the space), starting the search at the current line.

```
:L*TO ƀ
8 TO BE PRINTED ON THE TERMINAL
:
```

If the space were omitted, DED might find lines starting with "TOTAL", "TOO", "TODAY", etc.

A.2.3.3 TYPE LINES CONTAINING SPECIFIED STRING

Syntax: {#}F{n}.stringƀ

where a carriage return may be used in place of the period.

Description: Search for the specified string of characters anywhere in the n consecutive lines starting with the specified line, and type out the lines that contain string. The line pointer is positioned to the last line searched. If n is omitted, all lines are searched from the specified line to the end of working storage and the line pointer is left at the last line.

Example: Search Lines 1 through 5 for the string "OR " (note the space), and type out those lines containing the string.

```
:1F5.*OR ƀ
3 ONE OR MORE LINES
4 OR THE CURRENT LINE NUMBER
:
```

If the space were omitted, DED might find lines containing "ORDER", etc. Note, however, that with the space included, DED will not find lines containing "OR," or "OR:" or ending with "OR".

A.2.4 FILE COMMANDS

The file commands cause DED to copy a file into the working storage area, and to write the contents of working storage to a specified output file.

A.2.4.1 READ A SPECIFIED FILE

Syntax: {#}R

After "R" is typed, DED requests a filename; see example below.

Description: Insert a copy of the contents of the specified file into working storage after the specified line. The line pointer is positioned to the last line inserted.

Example: Read the file named MYFILE into working storage after the current line (Line 9).

```
:=9
:R
INPUT FILE: MYFILE.DEDb̸[OLD VERSION]b̸
238 LINES READ
:=247
:
```

A.2.4.2 WRITE A SPECIFIED FILE

Syntax: W

After "W" is typed, DED requests a filename; see example below.

Description: Copy all of the text in working storage to the specified file, replacing the content of the specified file. The line pointer is unaffected.

Example: Copy the text in working storage to a file named SOURCE.SUBRS.

```
:W
OUTPUTFILE: SOURCE.SUBRSb̸[NEW FILE]b̸
65 LINES WRITTEN
:
```

A.2.4.3 WRITE A SPECIFIED FILE AND QUIT

Syntax: Q

After "Q" is typed, DED requests a filename; see example below.

Description: Copy all of the text in working storage to the specified file, and then terminate DED.

Example: Copy the text in working storage to a file named SOURCE.SUBRS, and terminate DED.

```
:Q
OUTPUTFILE: SOURCE.SUBRS↵[NEW VERSION]↵
75 LINES WRITTEN
!
```

A.2.4.4 WRITE BACKUP FILE

Syntax: B

Description: Copy all of the text in working storage into a new version of the last file read (with an "R" command). If no file has been read, store the text into a new version of file BACKUP.DED. (If file BACKUP.DED does not already exist, it is created automatically by DED.)

A.2.5 MISCELLANEOUS COMMANDS

A.2.5.1 EXIT FROM DED WITHOUT WRITING FILE

Syntax: E

Description: Exit from DED without writing an output file. When this command is given, DED responds with the message [CONFIRM]. To exit from DED, the user must enter a carriage return. Any other character typed in will be taken as the first character of a new command, and DED continues.

A.2.5.2 TYPE COMMAND SUMMARY

Syntax: ?

Description: Type out a list of all DED commands with a brief description of their functions. This command can be terminated by typing ↑O.

A.2.5.3 OPTIONS

Syntax: On (Note: Letter "O", not zero.)
where n specifies an option. The option is set by the value of n, as described below.

Description:

n=1	Suppresses the typing of line numbers when lines are typed out by DED.
n=2	Suppresses prompt characters.
n=4	Causes each uppercase letter entered as text to be shifted to lower case, unless immediately preceded by ↑S; an uppercase letter can be entered by preceding it with a ↑S. This feature permits upper/lower case input from terminals that have only one case for letters.
n=8	Suppresses all bells.
n=16	Read 8-bit ASCII file (default is 6-bit).
n=32	Write 8-bit ASCII file (default is 6-bit).
n=0	Resets all options to their normal status.

NOTE: Combinations of the above options may be obtained by adding the numbers — for example, O3 would suppress both line numbers and prompt characters (since 3 = 1+2).

It is not normally necessary to use O16 or O32.

A.2.5.4 HELP

Syntax: H

Description: This command transfers from the DED subsystem to the HELP subsystem, initializing HELP with a special data base that enables it to answer questions about DED. Instructions on the use of HELP will be immediately typed out, followed by a question mark (?) which is the HELP prompt character. The user may then type in a question, terminated with either a carriage return or a question mark; HELP will type an answer and await another question. To exit from HELP, type END followed by a carriage return. This causes a transfer back to DED, with everything as it was before HELP was called. DED will type a colon (:) prompt and await another normal mode command.

SECTION A.3

CHARACTER EDITING MODE: COMMAND DESCRIPTIONS

It is important to bear in mind that although a few of the character editing commands have the same one-letter mnemonics as the commands used in normal mode, and sometimes perform analogous functions at the character-editing level, they are in fact different commands operating in a different mode.

The character editing commands are also unlike the normal commands in that they are not entered in response to a prompt and are not echoed on the terminal.

The basic text-editing functions — insertion, deletion, and replacement — are not performed directly upon the line being edited. Instead, the following three-stage process occurs.

(1) In response to the SET CHARACTER EDITING MODE command,

```
{#}C
```

DED types out the specified line and goes into character editing mode, awaiting character editing commands to edit the specified line. This line is henceforth referred to as the "old" line.

(2) In response to the user's character editing commands, DED constructs a <u>new</u> line, which is typed out character by character just below the old line as it is constructed.

(3) In response to one of three specific commands, DED replaces the old line with the new one in working storage and terminates character editing mode. A colon prompt is typed out and DED awaits a new command in normal mode.

The character editing commands permit the user to construct the new line as a modified copy of the old line by copying portions of the old line to the new line, skipping past other portions of the old line, inserting characters from the keyboard into the new line, etc. Thus when the old line is replaced by the completed new line, the effect is the same as a direct edit of the old line.

A "character pointer" is maintained during the process, which always points to a character in the old line; this character is called the "current character." Initially, the character pointer is positioned at the first character in the old line, and is then moved from left to right in response to various commands. When the character pointer is advanced past the last character of the old line, it is "off the line" and remains so. When the character pointer is "off the line" there is no current character.

The following is a capsule description of the basic character editing functions provided. The actual commands perform these functions and various combinations of them.

(1) Copy one or more characters from the old line (always starting at the current character) to the new line. The characters are appended one by one in sequence to the end of the new line. The character pointer is advanced to the character following the last character copied. This is the basic process of "constructing the new line from the old."

(2) Advance the character pointer in the old line without affecting the new line. This has the effect of skipping over characters in the old line; the skipped characters are not copied to the new line and may therefore be thought of as being deleted.

(3) Append input characters from the keyboard to the end of the new line. This accomplishes the effect of insertion.

(4) A combination of 2 and 3 above accomplishes replacement of characters.

(5) Replace the old line with the new line in working storage and terminate character editing mode.

The following sections describe the complete set of character editing commands in detail.

A.3.1 COPY CHARACTER FROM OLD LINE TO NEW LINE

Syntax: space

Description: When a space is typed by the user, the current character is copied to the new line. The character pointer is advanced to the next character in the old line. (If the current character is the last character on the old line, the character pointer is advanced "off the line.")

A.3.2 COPY UP TO A SPECIFIED CHARACTER IN OLD LINE

Syntax: Fa
where _a_ may be any character.

Description: When an F followed by any character is typed by the user, DED copies into the new line the sequence from the current character up to (but not including) the first occurrence of _a_ in the old line. The character pointer is left pointing at _a_. If _a_ is not found, all characters from the current character to the end of the old line are copied and the character pointer is left "off the line."

A.3.3 COPY UNTIL END OF OLD LINE

Syntax: E

Description: When an E is typed by the user, DED copies into the new line all characters from the current character through the last character in the old line. The character pointer is left "off the line."

A.3.4 "DELETE" (SKIP) ONE CHARACTER IN OLD LINE

Syntax: D

Description: When a D is typed by the user, the character pointer is advanced one character to the right in the old line. The new line is not affected. Thus the result is that a character in the old line is omitted or "deleted" from the new line.

A.3.5 INSERT TEXT FROM KEYBOARD TO NEW LINE

Syntax: I

Description: When an I is typed by the user, subsequent input characters from the keyboard are appended one by one to the end of the new line, until the user types a ↑Z, line feed, or carriage return. The position of the character pointer in the old line is not changed during this process, so the effect is that the input characters from the keyboard are "inserted" in front of the current character. Various control characters may be used to edit the string of characters during typein (see Section A.4).

If the input string of characters is terminated with ↑Z, the next character from the keyboard will be interpreted as a new character editing command. If the input string is terminated with a line feed, DED will immediately replace the old line with the new one and terminate character editing mode. If the input string is terminated with a carriage return, DED will copy into the new line all characters from the current character to the end of the old line, and terminate character editing mode.

A.3.6 "REPLACE" ONE CHARACTER

Syntax: Ra

where _a_ may be any character.

Description: When the user types an R followed by any character _a_, the character _a_ is appended to the new line and the character pointer is advanced to the next character in the old line. Thus the effect is to "replace" the current character with _a_.

A.3.7 TERMINATE WITH EXISTING NEW LINE

Syntax: line feed

Description: When the user types a line feed, DED immediately replaces the old line in working storage with the new line as it stands, and terminates character editing mode. DED then awaits a new command in normal mode.

A.3.8 COPY REMAINDER OF OLD LINE AND TERMINATE

Syntax: carriage return

Description: When the user types a carriage return, DED copies into the new line all characters from the current character to the end of the old line, replaces the old line in working storage with the resulting new line, and terminates character editing mode. DED then awaits a new command in normal mode.

A.3.9 ABORT

Syntax: X

Description: When the user types an X, DED throws away the new line and terminates character editing mode, leaving the old line unaffected in working storage. DED then awaits a new command in normal mode.

A.3.10 NOTE

When the character pointer has been advanced "off the line" (i.e., past the last character of the old line), only the I, X, line feed, and carriage return commands can be used. The effect of an I command in this situation is simply to append the subsequent sequence of typed characters to the end of the new line; the effects of line feed and carriage return are identical, i.e., replace the old line with the new line and terminate character editing mode.

When the character pointer is "off the line," any command except I, X, line feed, or carriage return will have no effect.

SECTION A.4

CONTROL CHARACTERS

Control characters are input by depressing the CONTROL key and holding it down while depressing and releasing some other key; for example, to type in a control Q, hold down the CONTROL key and strike the Q key. Throughout this guide, the notation ↑a (where a is any character) means "control a".

A.4.1 USE OF ↑Q TO ABORT A PARTIALLY TYPED NORMAL MODE COMMAND

When the user has partially typed a normal mode command and wishes to cancel it, he may do so by typing a ↑Q. Subject to the limitations given below, this will cause DED to abort the command, perform a carriage return, and type a colon (:) prompt to show that it is ready for a new normal mode command.

Limitations: If the ↑Q is entered while the user is typing the string portion of a SEARCH or SUBSTITUTE command (J, F, L, S, or Z), it does not abort the command; instead, it deletes the portion of the string that has already been typed in as explained below in Section A.4.2.

If a line number has already been typed in when the ↑Q is used to abort the command, the line pointer will be set to the specified line, although no other portion of the command is carried out.

A.4.2 USE OF CONTROL CHARACTERS TO EDIT TEXT DURING TYPEIN

When text is being typed in, either as lines of text under an APPEND or INSERT command in normal mode or as a string in a SEARCH or SUBSTITUTE command (J, F, L, S, or Z), the following control characters may be used to edit the text while it is being typed in. In addition, some of these control characters may be used when typing in text characters under an I command in character editing mode.

↑A	Delete last character typed in. Echoes "\" followed by the character deleted. May be used repeatedly to delete a sequence of characters from right to left. If the first character of a line or <u>string</u> is deleted, echoes a leftarrow (←) followed by a carriage return and an asterisk (*) prompt, exactly as if ↑Q (see below) had been used to delete the entire line or <u>string</u>. ↑A may be used under an I command in character editing mode. In addition, ↑A may be used while typing in a <u>filename</u> in a FILE command. In this case, the restriction is that ↑A cannot delete past the beginning of one of the fields in a <u>filename</u>; e.g., if the user is typing the <u>extension</u> portion of a <u>filename</u>, ↑A can be used to delete back to the period separating the <u>name</u> from the <u>extension</u>, but no farther.
↑Q	When a line of text is being typed in under an APPEND or INSERT command, ↑Q deletes the line (but does not delete any previous lines). When a string is being typed in under a SEARCH or SUBSTITUTE command, ↑Q deletes back to the beginning of the <u>string</u>. In either case, ↑Q echoes a left-arrow (←) followed by a carriage return and a colon (:) prompt, and the user may start over on the line or <u>string</u>. ↑Q cannot be used in character editing mode.
↑R	This causes DED to retype the line or <u>string</u> being typed in, i.e., perform a carriage return and type out the part of the line or <u>string</u> that the user has already typed in. The user may then continue where he left off typing in. This feature is useful when ↑A has been used repeatedly and the line or <u>string</u> has become hard to read. ↑R cannot be used in character editing mode.
↑S	This causes the next character typed in to be stored as an uppercase letter (if it is a letter). This is used for entering uppercase characters when input characters are being forced into lowercase as the result of an 04 command (see Section A.2.5.3). Thus if your terminal has only one alphabetic case, you can enter both upper and lowercase letters by using 04 to force lowercase, then using ↑S to enter

capital letters. If the next character typed in after a ↑S is not a letter, the ↑S has no effect. ↑S may be used under an I command in character editing mode.

↑Z This causes termination of text input when typing in a string, a sequence of lines under an APPEND or INSERT command in normal mode, or a sequence of characters under an I command in character editing mode.

A.4.3 CONTROL CHARACTERS FOR TAB, LINE FEED, ETC.

The following control characters may be used for special terminal functions on terminals that do not have the corresponding special function key — e.g., on a terminal that has no TAB key, ↑I may be used as explained below.

↑G	BELL.
↑I	TAB. DED has preset tab positions every eight spaces along the output line. The effect of a TAB or ↑I is to space over to the next tab position.
↑J	LINE FEED. Typing a ↑J is equivalent to striking a LINE FEED key.
↑L	FORM FEED. Typing a ↑L is equivalent to striking a FORM FEED key.
↑M	CARRIAGE RETURN. Typing a ↑M is equivalent to striking a CARRIAGE RETURN key.

CAUTION: Various terminals have their peculiarities, and a given type of terminal may not conform to the above description. For example, on a TI 700 Series terminal, ↑M is not equivalent to CARRIAGE RETURN, but merely echoes "↑]". The user is advised to experiment with his terminal before assuming that the above information is correct.

A.4.4 ↑C, ↑T, AND ↑O

These three control characters have special effects that do not depend on what the user is doing when he types one of them in.

↑C Causes DED to be terminated. The contents of working storage are not written out to a file, and are lost.

↑T Causes the system to type out a message on system status.

↑O If typed in while DED is typing out text or writing out a file, causes the output to be terminated prematurely. If DED is not in the midst of an output process, ↑O has no effect.

A.4.5 ENTERING CONTROL CHARACTERS AS TEXT

The control characters described in the above sections are not entered literally into a line or string when typed in, but instead cause the various special actions described above.

In addition, if any other control character is typed in, it will be echoed but will not be entered into the line or string being typed in.

The following methods may be used to enter a control character into the line or string being typed in, in the same way as any ordinary character.

A.4.5.1 ↑V

Any control character except ↑C, ↑T, or ↑O can be entered as an ordinary text character by preceding it with a ↑V. The ↑V is not stored as a text character, but causes the immediately following control character to be stored in the line or string as if it were an ordinary character. A ↑C, ↑T, or ↑O is not affected by the preceding ↑V, and will cause the effect described in Section A.4.4 when it is typed in.

A.4.5.2 ↑↑ (CONTROL UPARROW)

Any control character including ↑C, ↑T, and ↑O can be entered as an ordinary text character by the following procedure:

First type in a ↑↑ (control uparrow; see "Note" below). Then type in the ordinary character corresponding to the control character you wish to have stored in the line or string being typed in. The ↑↑ causes the immediately following character to be converted to the corresponding control character and stored in the line or string.

For example, to cause a ↑C to be stored as part of a line being typed in, first type ↑↑ and then type the letter C. The C will be converted to a ↑C and stored in the line.

Note: The ASCII character ↑↑ (control uparrow) cannot always be input by holding down the CONTROL key while striking the ↑ key; the procedure for inputting this special character depends on the particular type of terminal being used. On a Texas Instruments TI 700 Series terminal, hold down the CONTROL key while striking the period key.

SECTION A.5

COMMAND SUMMARY BY FUNCTION

A.5.1 TYPEOUT COMMANDS (NORMAL MODE)

Syntax	Description	Line Pointer Left at:
=	Type out number of current line.	Unchanged.
{#}/	Type out specified line.	Specified line.
↑	Type out line preceding current line.	Line preceding current line.
LF (line feed)	Type out line following current line.	Line following current line.
$	Type out number of last line in working storage.	Last line in working storage.
{#}Tn↵	Type out _n_ lines beginning with specified line.	Last line typed.
P	Type out all lines in working storage.	Unchanged.

A.5.2 EDITING COMMANDS (NORMAL MODE)

Syntax	Description	Line Pointer Left at:
{#}D	Delete specified line.	Line following deleted line.
{#}Kn↵	Delete _n_ consecutive lines, starting with specified line.	Line following last deleted line.
{#}I	Insert one or more new lines before specified line.	Specified line.
A	Append one or more new lines after last line in working storage.	Last new line appended.

Syntax	Description	Line Pointer Left at:
{#}M{n}.{m}b̸	Copy n lines starting with specified line, inserting them above line m.	Specified line.
{#}C	Enter character editing mode (see Section A.5.6) to edit text of specified line.	Specified line.
{#}S{string1}↑Zstring2b̸	Substitute string1 for first occurrence of string2 in specified line.	Specified line.
{#}Z{n}.{string1}↑Zstring2b̸	Substitute string1 for each occurrence of string2 within the n lines starting with the specified line.	Specified line.

A.5.3 SEARCH COMMANDS (NORMAL MODE)

Syntax	Description	Line Pointer Left at:
{#}Jstringb̸	Starting with specified line, find next line that starts with string. Move line pointer to this line and type it out.	Line found.
{#}Lstringb̸	Starting with specified line, find all lines starting with string and type them out.	Specified line.
{#}F{n}.stringb̸	Starting with specified line, find all lines within the next n lines that contain string and type them out.	Last line searched.

A.5.4 FILE COMMANDS (NORMAL MODE)

Syntax	Description	Line Pointer Left at:
{#}R	Copy contents of a specified file into working storage after the specified line.	Last new line inserted.
W	Write out working storage to a specified file.	Unchanged.

Syntax	Description	Line Pointer Left at:
Q	Write out working storage to a specified file and terminate DED.	n/a
B	Write out working storage to a new version of last file read; if no file has been read, write out to file BACKUP.DED (new version).	Unchanged.

A.5.5 MISCELLANEOUS COMMANDS (NORMAL MODE)

Syntax	Description	Line Pointer Left at:
E	Terminate DED.	n/a
?	Type out a summary of DED commands.	Unchanged.
On	Set options according to the following values of n: 0 Reset all options to default state. 1 Suppress typeout of line numbers. 2 Suppress prompt characters. 4 Shift all input characters to lower case. 8 Suppress all bells. 16 Read 8-bit ASCII file. 32 Write 8-bit ASCII file.	Unchanged.
H	Transfer to HELP. On exit from HELP, return to DED with everything unchanged.	Unchanged.

A.5.6 CHARACTER EDITING COMMANDS (CHARACTER EDITING MODE)

Syntax	Description	Character Pointer Left at:
space	Copy current character from old line to new line.	Next character.
Fa	(a = any character) — Copy characters from old line to new line, starting with current character and continuing up to (but not including) first occurrence of a.	First occurrence of a.
E	Copy characters from old line to new line, starting with current character, and continuing to end of old line.	"Off the line."
D	Advance character pointer without affecting new line, effectively deleting current character.	Next character.
I	Append subsequent input characters to new line, without moving character pointer. Sequence of input characters is terminated with ↑Z, LF, or CR; LF and CR have the effects described below.	Unaffected.
Ra	(a = any character) — Append a to end of new line and advance character pointer, effectively replacing one character of old line with a.	Next character.
line feed (LF)	Replace old line with new line, and terminate character editing mode.	n/a
carriage return (CR)	Copy characters from old line to new line, starting from current character and continuing to end of old line (same action as "E"); then replace old line with new line and terminate character editing mode.	n/a
X	Abort edit — throw away new line, leave old line unaffected, and terminate character editing mode.	n/a

APPENDIX B

GLOSSARY

APPENDIX B

GLOSSARY

The following terms are defined according to their specialized use in this guide. The vocabulary reflects the use of language in an environment influenced by the ILLIAC IV and its associated hardware and software.

The glossary makes detailed reference to sections in the guide, and may therefore be used as an index.

GLOSSARY

ACL — A Control Language for user interaction with the ILLIAC IV System. ACL consists of a set of statements; each ACL statement is a call to a subsystem that executes a selected function for the user. See *ACL executive, ACL statement, ACL subsystem, subsystem call,* and Sections 3 and 6.

ACL executive — the program resident in the ILLIAC IV System that processes ACL statements. The ACL executive takes ACL statements as input and initiates the subsystems called by these statements. See *ACL statement, ACL subsystem.*

ACL statement — a user statement that is passed to the ACL executive and causes initiation of an ACL subsystem. An ACL statement is thus effectively a subsystem call. See *ACL subsystem, ACL executive, ACL, subsystem call,* and Sections 3 and 6.

ACL subsystem — a program residing in the ILLIAC IV System that can be called by the user (using an ACL statement) to perform a specific function. See *subsystem call, ACL statement,* and Section 3.

active file — a file stored in a user's assigned active file space and listed in his file directory. Active files can be accessed by ACL subsystems under user control. See *archived file, directory, UNICON,* and Sections 4.4.1 and 4.4.2.

actual argument — information supplied by the user in an ACL statement or macro-call, in place of a formal argument in the statement format representation or macro definition. See *formal argument.*

archived file — a file that has been copied from active file space to the UNICON, and deleted from active file space. An archived file can be restored to active file space, but cannot be accessed by

ACL subsystems unless this is done. See *active file, directory, UNICON,* and Sections 4.4.1 and 4.4.2

area — a named, ordered collection of pages on the I4DM. An area is declared, named, and described in a FORMAT statement within a MAP subsystem call, and then assigned physical space on the I4DM by the ALLOC subsystem. See *areaname, FORMAT statement, I4DM, layout, logical disk,* and Sections 5.3 and 5.4.

areaname — the name of an I4DM area. The areaname is initially assigned by the user in the FORMAT statement used to declare, name, and describe the area (as part of a MAP subsystem call). After the area has been assigned space on the I4DM by the ALLOC subsystem, the areaname points to the first page of the area. An areaname consists of up to six alphanumeric characters, and must begin with a letter. See *area, FORMAT statement,* and Sections 5.3 and 5.4.

argument — variable information passed to a subsystem as part of an ACL statement or a subsystem control statement, or to a macro as part of a macro-call. See Sections 3 and 6.2.

Array Memory — working storage for the ILLIAC IV Processor. Array Memory consists of 128K (64-bit) or 256K (32-bit) words, with a 313-nanosecond cycle time. Each processing element in the ILLIAC IV Processor has access to one 2K (64-bit) or 4K (32-bit) block of Array Memory; each of these blocks is called a Processing Element Memory (PEM). The Control Unit has access to all of Array Memory. See *ILLIAC IV, Processing Element Memory,* and Sections 2.3.2 and 5.2.1.

ASK — the assembly language for the ILLIAC IV; also the assembler for this language, which resides as an ACL subsystem in the ILLIAC IV System. See Section 7 (ASK and GLYP).

batch job — a job whose primary input source is a file of ACL statements processed via a SUBMIT statement. The SUBMIT statement places in the batch queue a request to process the file; processing does not occur until the request reaches the top of the queue. See *job, batch queue,* and Sections 4.6 and 8.

batch queue — a system-maintained queue containing requests to process batch jobs. See *job, batch job,* and Section 8.

central file system — the set of facilities within the ILLIAC IV System used to store and manage user files. The term is also used loosely for the storage space used for files. See Sections 2.2.3 and 5.1.

control parameter — a named parameter with a signed decimal integer value. Control parameters are global to a job (see *job*) and the user can define them and assign values to them by means of EQU statements. One special control parameter named STATE is maintained by the system for each job; STATE always has a value assigned to it by the ACL subsystem called most recently within the job. This value is used to represent information about the execution (or failure) of the subsystem. Control parameters are used as elements in expressions. See *element, expression,* and Section 6.2.4.

control statement — see *subsystem control statement.*

Control Unit — the Control Unit of the ILLIAC IV Processor. See *ILLIAC IV Processor, Processing Element, ILLIAC IV,* and Section 2.3.2.

CU — see *Control Unit.*

directory — a named collection of information on a particular set of active files. The name is called a directoryname. One directory is assigned to each user, to contain information on active files "owned" by the user. The directoryname of the user's directory is the same as the user's userid. The DIR subsystem permits the user to obtain listings of one or more of the filenames of files in his directory, with the file size and date last written for each file. UDIR permits the user to obtain information on files in his directory that have been copied to the UNICON. Most ACL subsystems permit the user to access only files in his own directory; the COPY subsystem permits the user to create a copy (in his own directory) of a file in another directory. See *active file*, *userid*, *central file system*, *directoryname*, Section 7 (DIR and UDIR), and Section 3.4.2.1.

directoryname — the name associated with a particular collection of files (i.e., a directory) in the central file system. If the directory is a user's directory, the directoryname is the same as the user's userid. A directoryname enclosed in angle brackets is used optionally as the first field in a filename; if the user is referring to a file in his own directory, the directoryname and the angle brackets may be omitted and the user's userid will be used as the default directoryname for the filename. See *directory*, *filename*, *userid*, and Section 3.4.2.1.

element — part of an expression. An element may be a value (signed decimal integer), the name of a previously defined control parameter, or an expression enclosed in square brackets. See *expression*, *control parameter*, and Section 6.2.5.

entry point — a label within an ASK program that is declared to the assembler as an external entry point for the program. See Section 7 (LIB and LINKED).

expression — a sequence of elements and operators that can be evaluated as a signed decimal integer by the TEST and EQU subsystems. See *element*, *operator*, and Section 6.2.5.

extension — an optional portion of a filename. See *filename* and Section 3.4.2.1.

file — a named, ordered collection of information associated with a particular directoryname. Within the ILLIAC IV System files are held and managed by the central file system and may be deleted by means of the DEL statement and transferred by means of the COPY, MOVE, and CPYNET statements. Files may be edited interactively or listed on the user's terminal by means of the DED text-editor subsystem; additionally, DED may be conveniently used to interactively create files of alphanumeric text (such as files of ACL statements). A file is uniquely identified within a user's set of files by its filename. See *filename, directory, central file system, active file, archived file,* and Section 2.2.3.

file directory — see *directory.*

filename — the user identification of a particular file. A filename is made up of an optional directoryname, a name, and an optional extension and version; the directoryname is enclosed in angle brackets, the name and extension are separated by a period, and the extension and version are separated by a semicolon. See *directoryname, name, extension, version,* and Section 3.4.2.1.

formal argument — (1) a name used in a macro definition as a placeholder for information that will be included in the macro-call as an actual argument. In macro expansion, all formal arguments found in the macro definition are replaced by actual arguments supplied by the user in the macro-call. See Section 7 (MACRO). (2) a name of a class of arguments for ACL subsystems, used in the format representations in this guide to indicate information to be supplied by the user. See Sections 3 and 6.

formatspec — the portion of a FORMAT statement in a MAP subsystem call that describes the arrangement of one area on the logical disk. A formatspec is made up of operators separated by commas. See *operator, FORMAT statement, layout, area, logical disk,* and Sections 5.3.1.1 and 7 (MAP).

FORMAT statement — a control statement for the MAP subsystem. Each FORMAT statement declares, names, and describes one area. See *area, areaname, formatspec, layout, logical disk,* and Sections 5.3.1 and 7 (MAP).

GLYPNIR — a high-level language for writing programs to be executed on the ILLIAC IV. Also the GLYPNIR compiler, resident as an ACL subsystem in the ILLIAC IV System. GLYPNIR source code is compiled to ASK source code, which can then be assembled, link-edited, and transferred into Array Memory for execution. See *ASK* and Section 7 (GLYP).

Host — a computer system forming one node in the ARPA Network.

hostid — the unique identification of a Host in the ARPA Network. Either the Network name of the Host, or a corresponding octal number.

ILLIAC IV — the ILLIAC IV computer, as distinguished from the rest of the ILLIAC IV System. The ILLIAC IV includes the ILLIAC IV Processor, the Array Memory, and the I4DM. See Section 2.3.2.

ILLIAC IV Processor — the data processing components of the ILLIAC IV, comparable in function to the CPU of a conventional computer. The ILLIAC IV Processor consists of a Control Unit (CU) and 64 individual Processing Elements (PE's). The Processor is capable of parallel operation on 64 data streams. See *ILLIAC IV* and Section 2.3.2.

ILLIAC IV SAVE file — a file containing a complete ILLIAC IV memory image, together with loader instructions to set the initial machine state. ILLIAC IV SAVE files are produced by the LINKED subsystem for input to the RUN or SSK subsystems; in addition, the dump files produced by RUN and SSK are ILLIAC IV SAVE files. The term "ILLIAC IV SAVE file" is commonly abbreviated to "ISV file," and the characters ISV are used as the default extension in the filename of an ISV file. See Sections 4 and 7 (LINKED, RUN, and SSK).

ILLIAC IV System — the system discussed in this guide.

interactive job — a job whose primary input source is a sequence of ACL statements typed in at the user's terminal. These statements are transmitted directly to the ACL executive and are processed immediately. See *job* and Section 8.

ISV file — see *ILLIAC IV SAVE file*.

I4DM — the ILLIAC IV Disk Memory, which serves as the main memory store for the ILLIAC IV. See Section 5.2.

job — the processing of a sequence of ACL statements from a primary input source. In an interactive job, the primary input source is the user's terminal, and the job begins with the user's LOGIN and ends with his LOGOUT. In a batch job, the primary input source is a file of statements (i.e., a PIF) named in a SUBMIT statement in another job. The batch job begins with the entry of a request into the batch queue as a result of the SUBMIT statement. The processing of a batch job is the processing of the ACL statements contained in the PIF (and in any alternate input sources) and ends with a LOGOUT statement or the EOF in the PIF. See *batch job, interactive job, jobid, primary input file, primary output file,* and Sections 4 and 8.

jobid — a unique identifier assigned by the system to each job. The jobid is generated by the system when the job begins. The jobid must be entered by the user in DELJOB and INQ statements. See *job* and Section 8.

label — (1) the name of an ACL statement. Labeled ACL statements can be branched to under control of the TEST statement. See Sections 6.1, 7 (TEST), and 8. (2) the name of an ASK instruction.

layout — the relative arrangement of one or more I4DM areas. See *area, FORMAT statement, formatspec, logical disk, I4DM,* and Section 5.3.

library — a collection of files of relocatable ASK object code, together with a library file created by the LIB subsystem that lists all files in the library and all the entry points within the files. A specified library file can be searched by the link editor to resolve external references in a program being link-edited, and the corresponding files of code will be link-edited into the program. See Section 7 (LIB and LINKED).

library file — see *library*.

LINKED — the link-editor subsystem in the ILLIAC IV System. LINKED takes files of relocatable ASK object code as input, and creates an ILLIAC IV SAVE file for the ILLIAC IV or SSK. See *ILLIAC IV SAVE file* and Section 7 (LINKED).

link editor — see *LINKED*.

logical disk — the logical model of the I4DM presented to the user by the I4DM utility software, for purposes of describing I4DM layouts. See *area, I4DM, formatspec, FORMAT statement, layout,* and Section 5.2.

name — the first portion of a filename. See *filename* and Section 3.4.2.1.

operator — (1) one of the items making up a formatspec. The formatspec operators have the forms ±nP, ±nS, ±nB, ±nR, ±nL, X, and T (where n is any integer). See Sections 5.3 and 7 (MAP). (2) one of the set of arithmetic, logical, shift, and unary symbols used in an expression. See Section 6.2.5.

page — a page is the smallest addressable unit of data words in a file or in an I4DM area. In an area, a page is a collection of 1024 (64-bit) or 2048 (32-bit) contiguous words on the I4DM.

parameter — see *control parameter*.

password — a unique alphanumeric string associated with each userid for security purposes.

PE — see *Processing Element*.

PEM — see *Processing Element Memory*.

PIF — see *primary input file.*

POF — see *primary output file.*

primary input file — a file of ACL statements used as the primary input source for a batch job. See *job, batch job,* and Sections 4 and 8.

primary output file — a file used for writing out system messages and responses resulting from a batch job (in an interactive job these messages and responses are sent to the user's terminal). See *job, batch job,* and Sections 4, 7 (SUBMIT), and 8.

Processing Element — one of the array of 64 identical arithmetic units in the ILLIAC IV Processor. The 64 Processing Elements (PE's) are controlled by the Control Unit, which decodes instructions and sends the same control signal to each PE in the array; all PE's then execute the same operation in parallel, each using data stored in its own portion of Array Memory. See Section 2.3.2.

Processing Element Memory — the portion of Array Memory that can be accessed by one of the 64 Processing Elements. Each PEM is 2K (64-bit) or 4K (32-bit) words. See *Array Memory, ILLIAC IV Processor, Processing Element,* and Section 2.3.2.

SSK — a subsystem in the ILLIAC IV System that simulates execution of a program on the ILLIAC IV, returning various performance timing statistics and information requested in DISPLAY statements in the user's ASK program. SSK uses the B6700 resource of the ILLIAC IV System. See Section 7 (SSK).

statement — (1) an ACL statement. (2) a subsystem control statement. See *ACL statement, subsystem control statement*, and Sections 3 and 6.

STATE parameter — a special control parameter defined for each job by the system and having a value assigned to it by the subsystem most recently executed within the job. See *control parameter* and Sections 6.2.4 and 7 (EQU and TEST).

subsystem — see *ACL subsystem.*

subsystem call — most subsystems are called by a single ACL statement, containing all the information necessary for the subsystem to execute. Certain subsystems (e.g., LINKED and MAP) require one or more control statements following the ACL statement to make up a complete call. Functionally, a subsystem call is a request to initiate processing of information contained in the call; processing is actually initiated by the ACL executive. See *ACL statement, ACL subsystem, ACL executive,* and Section 3.

subsystem control statement — a statement that may be entered as part of the calling sequence for a particular ACL subsystem. Examples are the SEG, SET, and INCLDE control statements for the LINKED subsystem, and the FORMAT, PRINT, and TIME control statements for the MAP subsystem. See *ACL subsystem, subsystem call,* and Section 3.5.

system — see *ILLIAC IV System.*

UNICON — the system's mass information storage resource. The UNICON uses a laser to write and read information on coated Mylar strips, and has a capacity of approximately 700 billion bits of on-line storage. The UNICON cannot be addressed or accessed by the user or by ACL subsystems under user control; however, files stored on the UNICON can be retrieved by the system operator as explained in Section 4.4.2. See Sections 2.3.3, 4.4.1, and 4.4.2.

userid — an identification name associated uniquely with a particular user of the ILLIAC IV System. userid's are administratively assigned, and are supplied by the user in the LOGIN and SUBMIT statements. See Section 8.1.1.

value — a signed decimal integer. In ACL usage, values may be assigned to control parameters by the user via EQU statements; and expressions used in TEST statements result in values when evaluated. See *expression*, *control parameter*, and Sections 6.2.4, 6.2.5, and 7 (EQU and TEST).

version — a number used as the last part of a filename to distinguish among files with the same name and extension. The version may be created and handled by the system without explicit reference by the user. See *filename* and Section 3.

APPENDIX C

BIBLIOGRAPHY

(To be supplied)