Global Development 16-bit File Management Manual Version 8.1

TABLE OF CONTENTS

APPENDICES

1. Introduction to File Management

1.1 Basic Concepts

Global Cobol provides a number of different language statements to allow you to define and manipulate files on different devices such as printers and direct access devices. The file definition statements, coded in the data division, include FD, ASSIGN, RECORD LENGTH, SIZE, OPTION, KEY, KEY LENGTH and BLOCK CONTAINS. They are used to specify the attributes of a file. The file processing statements, coded in the procedure division, allow you to manipulate data records. The statements are OPEN, WRITE NEXT, WRITE, REWRITE, READ and CLOSE. In addition a number of system routines are provided to perform other useful functions necessary in a complete file management system.

File handling is always the most difficult area of programming to understand since you have to be familiar with a large variety of concepts, most of which are peculiar to the particular operating system under which your program is to run, rather than the language you are writing in. Thus before examining the Global Cobol statements involved in more detail, a number of important underlying concepts must be explained. This is the purpose of this introduction.

1.1.1 Documentation Structure

Global file management features are documented to reflect the way in which the System Manager file processing is to some extent independent of the particular physical medium and file organisation involved.

Obviously it is not possible to read records at random from a printer, for example, but you can write a program to output a print file and then decide at run-time whether the file is to be assigned to a real printer or to a direct access device. Similarly, if your program only reads or writes a data file sequentially, you can assign it to any kind of direct access storage. The conventions for writing print files and files which can be assigned to direct access storage are described later in this chapter (in section 1.5), so as not to complicate the rest of the documentation with a lot of device-dependent detail.

Files may be structured in a number of different ways. For example, a file may be made up of consecutive fixed length records which can be accessed either sequentially, or at random, by supplying the record number as a key. Alternatively, an indexed sequential file consists of a data area, an index area and an overflow area. Each record contains a key, which might be a part code or a customer number for example, and a chaining technique is used to maintain the file in ascending key sequence. The file can be either scanned sequentially, or records can be retrieved at random according to their key. Records with new keys can be added to the overflow area and chained into the existing file so the correct sequence is maintained.

Entry point(s)	Subroutine Name (reference)	Function
CONV$	File Conversion (15.1)	Converts relative sequential files to indexed sequential format, and vice versa; re-organises indexed sequential files, removes logically deleted records.

COPY$	File copy (15.2)	Copies direct access files.
CATA$	Catalogue (15.3)	Allows programs to be essentially device independent by completing volume and unit information in a list of file definitions from data held in a catalogue file.

DELE$	Delete (15.4)	Deletes a named file from a direct access volume.
RENA$	Rename (15.5)	Renames a file on a direct access volume.
FILE$	File information (15.6)	Prompts the operator to supply the file and unit information for a particular file definition.
VOLID$	Volume identification (15.7)	Supplies the volume-id of the volume currently mounted on a direct access unit.
ASSIG$	Assignment (15.8)	Allows unit assignments to be made under program control, and the assignments currently in force to be examined. The routine can be used to determine the type of device to which a file is actually assigned.
OPEN$ OPENS$ LIST$ CLOSE$	Directory (15.9)	Scans the directory of a direct access volume and returns details of the file or files that it contains one by one.
FSTAT$	File Status (15.10)	Allows the program to determine the used file size.
SET$	Set Password (15.11)	Allows you to alter or set up the current password.
SECUR$	Secure File (15.12)	Secures a file on a direct access volume with the current password.

Table 1.1.1 - System Routines used for File Management

ENTRY POINT(S)	ROUTINE NAME (REFERENCE)	FUNCTION

CALC$	ISAM File size (15.13)	Calculates the size required for an indexed sequential file
FIX$	Fix number & date (15.4)	Fixes the Product Serial Number and the Expiry date for a file
SCR$	Scratch Volume (15.15)	Deletes all files from a diskette, direct access volume or subvolume.
DEVIN$	Device Information (15.16)	Provides complete information about a physical device (sector size, number of cylinders, etc etc).
ISUSE$	ISAM record in use (15.17)	Determines the number of records in use on an ISAM file.
SLOCK$ SULOC$	Shared Lock (15.18)	Obtains a shared lock on a particular file, used in master/servant record updating.
LWORK$ UWORK$	Lock Work Unit (15.19)	Obtains an exclusive lock on a work unit to avoid interference from competing processes.
OPDE$	Open file with optional delete (15.20)	Opens a file prompting the operator for deletion if required.
LIBR$	Build copy library (15.21)	Build an index for the supplied copy library.
GTDES$ PTDES$	Get volume description (15.22)	Retrieves the volume description for the specified volume.
SUBS$	Get sub-volume size (15.23)	Retrieves the size of a domain sub-volume and the maximum space available to allocate a new subvolume.

Table 1.1.1 - System Routines used for File Management (cont.)

The way in which a file is structured is known as the file organisation. The System Manager supports a number of different organisations.

To make file handling as simple as possible, Global Cobol uses common file processing statements, such as OPEN, READ, WRITE and CLOSE, to manipulate a file, irrespective of its organisation. (The particular organisation involved is actually indicated by information supplied in the file definition.) The normal way in which file definition and file processing statements are used is described in 1.2 and 1.3. However, there may be minor differences according to the actual file organisation you are using so each statement is documented again

independently in the chapter devoted to the organisation in question. To avoid too much unnecessary repetition, these detail chapters refer back to any relevant parts of the general description which follows.

Several organisations are available to allow you to access DOS or Unix files from within the System Manager environment. However the File Converters package is available to allow you to access these files more directly.

There are, in addition to the language statements, a number of system routines involved in file management. They are documented in detail in Chapter 9 of this manual. Table 1.1.1 summarises their functions.

1.1.2 Device-independent Programming

When the System Manager is installed the devices supported by a configuration are allocated unique 3 character unit addresses. For systems which are not networked and are not separated systems, all three characters are numeric. Thus by convention:

- units 100 - 199 are diskette drives (but 109 is always reserved for local RAM disk, if present);
- units 200 - 299 are hard disk subunits (see below);
- units 500 - 599 are printers;
- units 800 - 999 are direct access devices on another computer linked to your own computer via $REMOTE.

On networks local units are addressed in the same way, but additional addresses are available for accessing direct access units on other computers, each such file server being identified by a letter A - Z:

- units 600 - 699 are hard disk subunits on the master computer; these can also be accessed by means of the network address, e.g. X00 - X99 if X identifies the master computer;
- a00 - a99 are diskette drives on computer A (a09 being reserved for the RAM disk on computer A, if present);
- A00 - A99 are hard disk subunits on computer A.

Note that there are no network addresses for printers, which can only be accessed directly from the computer to which they are attached. If it is necessary to share printers on a network this must be done by means of a spool unit. Note in addition that the network form of address can also be used to access local units.

On separated systems the units are addressed in a similar way to networked systems but there are no local units and all units for each file server system are identified by a letter A - Z:

- units 600 - 699 are subunits on the master system; these can also be accessed by means of the system address, e.g. A00 - A99 if A where A is assumed to be the master system;
- a00 - a99 are diskette drives referred to by system A ;

- A00 - A99 are hard disk subunits referred to on system A.

Usually there is a single unit address corresponding to each physical device, but sometimes a single device may be described by a number of different addresses. For example, it may be decided to subdivide a large hard disk into a number of different subunits, each with their own unique unit address, in order to make it easier to allocate different applications their own private file space. Sometimes when the same diskette drive can hold different media (e.g. single or double density diskettes) the same device is given a specific unit address for each medium.

A Global Cobol program should normally be written to be independent of the actual physical unit addresses supported by a particular configuration, in order that it can run, unchanged, at a number of sites. This is achieved by associating each file used by the application with a "logical" unit-id rather than an actual unit address. The relationship between unit-ids and unit addresses is only fixed when the program comes to be run, by which time, of course, the addresses to be used are known. The unit-ids consist of up to three ASCII characters which, to prevent them being confused with unit addresses, should be neither entirely numeric nor an alphabetic character followed by 2 digits.

The process of associating unit-ids with unit addresses is known as assignment and is discussed in detail in the Global Operating Manual. The latest time at which an assignment can be made is when the file involved is opened, at which stage the operator will be prompted to supply the unit address corresponding to the unit-id if this is not already known. The assignment command program ($A) can be used to preset assignments at the start of the session to prevent prompts appearing when files are opened. In addition the $CUS Permanent Unit Assignments option can be used to fix those assignments which are permanently required at an installation, and programs to be run from the menu established by customising the menu file using the MN utility can have any necessary assignments made as part of that customisation.

The unit-ids that you employ in coding an application should consist of one, two or three ASCII characters, the first of which should be alphabetic, and must not be the $ sign. For example:

```
DA1  DA2  DA3  PR
```

Do not use unit-ids which consist of an alphabetic character followed by 2 digits, to avoid confusion with LAN unit addresses.

Unit-ids beginning with a $ sign are used by the System Manager itself. For example, $CP identifies the unit from which command programs are loaded, $P the unit for application programs, and $PR the logical printer. There is a list of the most commonly used System Manager unit-ids in the Operating Manual.

The unit-id consisting of a single ? character, followed by two blanks, has a special meaning when the CATA$ and FILE$ system routines are employed. You must not attempt to use it as a normal unit-id.

1.1.3 Volumes

A volume is a storage medium, such a diskette, hard disk or RAM disk, or a volume file in a separated system, which may occupy a unit and

hold one or more files. Each volume is identified by means of its volume-id, a six character ASCII code and may also have an associated fifty character volume description. Before a volume can be used for data storage you must run the System Manager volume maintenance command program ($V) to initialise it and write its volume-id (and possibly a volume description) to a magnetic label stored in a special part of the volume known as the directory.

The volume concept applies only to direct access devices. It does not apply to printers because there is no means of holding machine readable label information on these devices.

When you initialise a volume you should not normally give it a volume-id beginning with the characters SYS or BAC, since such volume-ids are used for the System Manager itself or by the Data Security System.

Volumes may be either fixed (e.g. hard disks) or exchangeable (e.g. diskettes). By careful choice of volume-ids you can guarantee that all exchangeable volumes are uniquely labelled. You can supply an optional clause in the file definition to instruct the System Manager to ensure that the correct volume is mounted before you use it.

1.1.4 Files

Before you access a file from your program you must "open" it using the Global Cobol OPEN statement. Each open file is expensive in terms of the System Manager resources and there are always limitations on the number of files which can be open simultaneously. These restrictions are either a characteristic of the physical device involved, or the System Manager software itself.

For example, only one file can be open at any one time on each printer because of the sequential nature of the medium. However, a direct access volume can contain a number of files and in theory all of them might be open simultaneously. Since each open file consumes buffer space within the System Manager nucleus, there is a limit on the number of direct access files that can be open at any one time. In practice, however, application programs which are to run on the largest possible number of different configurations are normally coded to require no more than 7 direct access files to be open at any one time. The restrictions you should bear in mind when coding portable programs are summarised in Appendix C of the Global Cobol Language Manual.

When you have finished working with a file you should "close" it using the Global Cobol CLOSE statement. This completes any outstanding I/O operations and frees any System Manager resources used in accessing the file. The file definition can then be re-used to process another file, should you so wish.

Each file is identified by means of its file-id, an 8 character ASCII code which is saved in a magnetic label when a direct access file is created. The file-id is ignored in the case of print files.

You should avoid creating application file-ids whose names contain the $ character, or which start with a prefix (a letter followed by a full stop), since such names are used by the System Manager itself.

Spool units are used for storing print-image and text files to be printed, and then finally deleted, by the $SP command. Any file

written to such a unit is automatically renamed by the System Manager so that the new file-id consists of the originating operator-id concatenated with a unique sequence number, in order that file name clashes cannot occur. Existing files on a spool unit are under the control of the System Manager Spooler and cannot be opened by ordinary programs.

A direct access device can hold a number of files, distinguished by their different file-ids. The maximum number of files that can be held on a direct access volume depends on the size of its directory. At least 50 files can be held on all devices supported under System Manager V6.0 and later systems, and most subunits (on hard disk) are capable of holding 99 files. To find the number of directory entries available for files on any particular type of volume you need only list it using the file utility ($F) LIS command described in the Operating and Utilities Manuals.

1.1.5 Records

A file is considered to be made up of a number of records. A record is created or replaced whenever the program executes a WRITE NEXT, WRITE, REWRITE or DELETE statement to output an area of memory to the file. Later a READ FIRST, READ LAST, READ NEXT, READ PRIOR or READ statement is used to retrieve a record from the file and bring its contents into memory for the program to work with.

The format and content of a particular record is mainly determined by application requirements. However, there are certain restrictions, normally affecting header information at the start of the record, which apply to print records, indexed sequential, text file, and variable length file records. There are also additional conventions to obey if records are to be processed by AutoClerk. Record formats are discussed in the chapters dealing with the particular file organisations affected, except for print file records, which are described in 1.5 below.

1.1.6 File Space Allocation

When a new direct access file is opened information supplied in the file definition tells the System Manager the amount of space to allocate to it. You may either make a specific request for, say, 20000 bytes, or alternatively ask that you be allocated the maximum contiguous space available. In either case if the open succeeds the file will be allocated a fixed contiguous area of storage termed its extent. Once the initial allocation has been made the file's extent cannot be increased by your program: if a larger extent is required you will need to copy the data to a new file which has been allocated more space, or use $REORG (described in the Utilities Manual) if the file is on hard disk.

I/O ERROR CONDITION	STATEMENT	POSSIBLE RECOVERY ACTION
Attempted to access an unsupported device due to an error in unit assignment	OPEN	None. The error is irrecoverable

Attempted to open a file which is already open and cannot be shared	OPEN	Wait for the other user(s) to finish with the file, then retry.	
Attempted to open more files simultaneously than the device is capable of supporting	OPEN	None. The error is irrecoverable.	
Attempted to allocate a new file on a volume with insufficient directory or data space	OPEN NEW	Either exchange the volume for another containing less files or more contiguous free space, or delete some file or files from the volume using another terminal. Then retry.	
Attempted an operation which is inappropriate for the physical device involved: e.g. a direct access READ, WRITE, READ NEXT or REWRITE on a printer	READ READ NEXT WRITE WRITE NEXT	None. The error is irrecoverable.	
Attempted to write to a protected file	REWRITE WRITE WRITE NEXT	Release the file from protection using another terminal. Then retry.	
Attempted to write to a drive or volume which is hardware write protected	REWRITE WRITE WRITE NEXT	Release the volume from protection as documented in your hardware manual. Then retry.	
Printer low on paper	WRITE NEXT	Mount and align new stationery, then retry to continue.	
Hardware read or write error	all	Retry a number of times in the hope the error will prove intermittent.	

Table 1.1.7 - I/O Error Conditions and Possible Recovery Actions

For volumes which have been initialised as spool or work volumes, there is a nominal maximum allocation size, usually much less than the true capacity of the volume. This nominal maximum determines the greatest amount of free space actually allocated when a program makes a non-specific space request for a new file. This means that a number of "maximum" size files can be open simultaneously, which is not generally possible on an ordinary unit where the first non-specific space request is liable to obtain all the free storage available. Note that files larger than the nominal maximum can still be allocated by making specific space requests. For files on spool volumes, if the initial allocation of space is used up, a further extension file is allocated automatically.

Files are typically created by writing records to consecutive storage at the beginning of the extent, so the extent contents can be pictured diagramatically as follows:

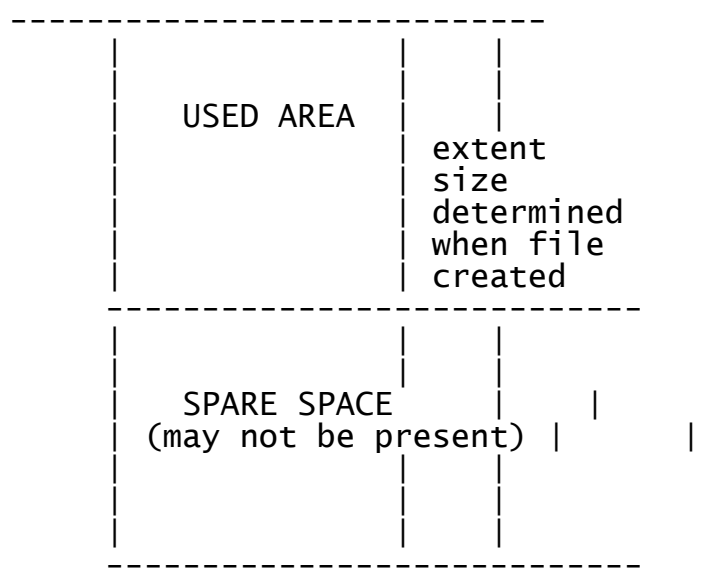


Figure 1.1.6 - File Space Allocation

The same diagram can be considered to apply to printers as well. The extent size may be determined from a specific request coded in the file definition, or may be defaulted to a maximum determined by the System Manager. In the case of a printer this is the largest positive number that can be held in a PIC 9(9) COMP field (more than 2 billion bytes) so that the default maximum is for all practical purposes infinite.

For direct access devices a special field, the write next pointer, is saved in the file label to address the current end of the file. The presence of this pointer allows an existing file with spare space available at the end of its extent to be extended with new records output by WRITE NEXT (hence the name given to the pointer).

The System Manager prevents the WRITE NEXT statement from creating records outside the file extent. The random access READ and WRITE statements and the sequential READ NEXT and READ PRIOR are normally only allowed to access records within the used area of the file.

For direct access files only, a special option of the CLOSE statement allows you to "truncate" the extent in order to eliminate the spare space, if any, and make the storage thus freed available for re-allocation.

1.1.7 I/O Error Handling

Normally, if an I/O error occurs on a printer or direct access device, The System Manager displays an explanatory console message together with the retry prompt, which gives the user the opportunity of either abandoning or retrying the current operation. Sometimes one or more retries will result in the problem being overcome and in this case the application will continue as though the error had never taken place. Table 1.1.7 shows the typical I/O error conditions that may affect the various file processing statements, and the possible recovery actions, if any. A program may, by means of the ON ERROR statement in the FD,

intercept I/O errors before any message is displayed by the System Manager, and if desired suppress the System Manager message.

The operator can usually retry a failing operation any number of times, or abandon the attempt at any stage by replying appropriately to the prompt. The System Manager will then signal that an irrecoverable I/O error has occurred. The affected job will then be terminated immediately unless a special option has been coded in the file definition to indicate that the program itself is prepared to handle irrecoverable I/O errors on that file.

There are three other conditions that are treated as irrecoverable I/O errors, since their effect is essentially the same as far as the application program is concerned, in as much as it is prevented from accessing data it requires. These conditions arise if the operator is unable to assign the file's logical unit-id to an appropriate physical unit address, or if he is unable to mount a volume with a specified volume-id, or if he is unable to supply the correct password for a password protected file. Only the OPEN statement can be affected.

For print files an irrecoverable I/O error is signalled if the operator is unable to supply special stationery when requested to do so by the forms control prompt output as a result of a WRITE NEXT statement, as explained in 1.5.

1.2 The File Definition

You must code a file definition (FD) in your program's data division for each file that is to be open at the same time. A number of different Global Cobol statements are used in constructing the file definition, which is a data area shared by the application program and the access method, the Global Cobol routine entered whenever a file processing statement - such as OPEN, READ, WRITE or CLOSE - is executed.

The file definition is normally coded in working storage. However, it is possible to pass a file definition as a parameter to a routine entered by means of a CALL or EXEC statement, and in such a routine the passed FD would be coded in the linkage section.

Some of the statements employed in creating a file definition vary according to the file's organisation and the access method used to process it. However, those described in detail below are common to all organisations and are explained here to prevent duplication in the chapters dealing with the organisations individually.

The general format of a file definition is:

```
FD filename ORGANISATION organisation
[ASSIGN TO UNIT unit-id FILE file-id [VOLUME volume-id]]
[RECORD LENGTH IS length]
[SIZE IS size]
[OPTION ERROR]
[ON ERROR intercept]
[other optional statements, depending on the organisation]
```

The FD statement must always begin the construct, and be followed by an ASSIGN statement if one is present. The other optional statements can be coded in any order.

The construct establishes a special group data item whose name is filename. The quantities unit-id, file-id, volume-id, length and size appear as subordinate elementary items within the group and can, if need be, be referred to by the application program.

If it is possible to specify unit-id, file-id or volume-id before the program executes then the quantity should be coded as a character string in double quotes. If you can specify the size or length before the program executes, code the quantity as a numeric string. If any of these quantities is not known until run-time then a symbol must be coded for the quantity. This symbol will then label a level 02 item which the user program is responsible for initialising.

1.2.1 The Filename

The filename must be a symbol, as defined in 2.1.2. It labels the file definition and must appear as the first operand of any file processing statement accessing it.

1.2.2 The Organisation

The ORGANISATION (or ORGANIZATION) clause determines which access method routine is to be included from the system library in order to process the file. The first two bytes of a file definition coded in working storage are set to address the entry point of the access method, so that file processing statements which reference the FD pass control to the appropriate routine. The American spelling ORGANIZATION is accepted as an alternative to the UK spelling ORGANISATION, but for consistency the UK spelling is used throughout the rest of this manual.

The different file organisations supported by the System Manager and Global Cobol are: relative sequential (RS); indexed sequential (IS); text file (TF); variable length record (VL); basic direct (BD); data library (DL) and physical sector (PS). Also described here is the Global Cobol access method for Speedbase files. Some other file organisations exist which are outside the scope of this manual, most notably DMAM (covered in the Data Management Manual) and the Finder and Planner access methods.

Relative sequential and indexed sequential files are most frequently used in commercial applications, and are described in the next two chapters of this manual. For them the ORGANISATION clause is coded as either:

```
     ORGANISATION RELATIVE-SEQUENTIAL
or:
     ORGANISATION INDEXED-SEQUENTIAL
```

For the other organisations, which are covered in subsequent chapters, you must code an ORGANISATION (or ORGANIZATION) statement whenever your program wishes to make use of them. You code:

```
     ORGANISATION name TYPE type EXTENSION size
or:
     ORGANIZATION name TYPE type EXTENSION size
```

where name is the name of the access method routine (and is automatically made a global symbol), type is an integer between 0 and 99 inclusive, and size is an integer between 0 and 1016 inclusive. The size must be a multiple of 8. The ORGANISATION statement can be coded

anywhere within the data division, provided it precedes any file definitions which specify it.

The American spelling, ORGANIZATION, may be used in place of the British form, ORGANISATION, as in the case of the ORGANISATION clause. You may also use the abbreviation ORG.

The ORGANISATION statement is a directive to the compiler which occupies no storage itself, but affects the way in which file definitions (FDs) are generated. The compiler relates an FD to a particular ORGANISATION statement by the name that appears both in the statement and the file definition's ORGANISATION clause.

For example, the ORGANISATION statement required for the variable length record access method is:

```
ORGANISATION OR$84 TYPE 4 EXTENSION 8
```

A typical variable length record file definition might then be:

```
FD VLFIL ORGANISATION OR$84
ASSIGN TO UNIT "DSK" FILE "OUTPUT"
```

Every FD generated contains a common prefix which is 80 bytes in length. The size specified in the EXTENSION clause is the length of the access method dependent part which follows the prefix.

The type field is generated within the FD prefix and stored in the file label whenever the access method creates a new file. Then whenever an existing file is opened the access method can check that it is of the right type. If not, the file not found condition is generated.

Note that although a number of ORGANISATION statements may be coded in the same program no two of them must have the same name. The documentation of each of the special access methods defined later in this manual includes a description of the particular ORGANISATION statement (and ORGANISATION clause) which must be coded whenever the access method is used.

You may also code the ORGANISATION clause as:

```
ORGANISATION UNDEFINED
```

in order to set up a file definition for use by certain system routines which process files without requiring an access method to be present. The result will be that an FD, 80 bytes in length, is expanded for the system routine to use, but no access method is included. The pointer at the start of the FD will then be set to -1 so that if a file processing statement is erroneously attempted your program will be terminated with an illegal jump program check. The documentation of the individual system routines indicates where ORGANISATION UNDEFINED may be used.

Only direct access devices support all types of file organisation. The restrictions applying to printers are summarised in section 1.5.

1.2.3 The ASSIGN Statement - General

The ASSIGN statement is required for every file definition coded in working storage and, in this case, it must be the statement

immediately following the FD statement. The ASSIGN statement may be omitted for file definitions appearing in the linkage section.

1.2.4 The ASSIGN Statement - UNIT Clause

The unit-id in the UNIT clause is coded as either "uuu", where uuu is one to three ASCII characters, or as a symbol. When a symbol is used the following statement is generated in the FD:

```
02 symbol PIC X(3)
```

The application program must place the unit-id in this field before opening the file.

Note that the System Manager treats a unit-id of ?, i.e. a question mark followed by two blanks, specially. Although you may code such a unit-id in the ASSIGN statement, you must have replaced it with a true identifier (by using the CATA$ routine or FILE$ routine described in Chapter 9) before attempting to open the FD. The System Manager will terminate any program which attempts to open a file assigned to unit ? with an error.

1.2.5 The ASSIGN Statement - FILE Clause

The file-id in the FILE clause is coded either as "ffffffff", where ffffffff is one to eight ASCII characters, or as a symbol. When a symbol is used the statement:

```
02 symbol PIC X(8)
```

is generated within the FD. The application program must place the file-id in this field before opening the file.

The file-id is ignored in the case of a printer. When an existing file is opened on a direct access device, a file condition (explained later) is signalled if a file with the specified file-id is not present on the unit. A file condition is also generated if you attempt to open a new file on a direct access volume which already contains a file with the same file-id as that coded in your FILE clause.

1.2.6 The ASSIGN Statement - VOLUME Clause

The VOLUME clause is optional. When present volume-id checking will be performed when the file is opened to ensure that the volume-id of the volume currently online matches that specified in the VOLUME clause. If there is a mismatch the System Manager will prompt the operator to mount the correct volume before returning control to the statement following the OPEN. The VOLUME clause is ignored in the case of a printer.

When a VOLUME clause is coded the volume-id must be specified as "vvvvvv", where vvvvvv is one to six ASCII characters, or as a symbol. When a symbol is used the statement:

```
02 symbol PIC X(6)
```

is generated within the FD. The application program must place the volume-id in this field before opening the file. By moving LOW-VALUES to the field the program is able to suppress volume-id checking as though the VOLUME clause had never been coded in the first place.

1.2.7 The RECORD LENGTH Statement

Although the RECORD LENGTH statement may be used in any file definition its precise meaning differs according to the file organisation involved.

For RS, IS and DL files it must be coded when a new file is being created, and specifies the size in bytes of the fixed length records that make up the file.

For VL, TF and some other organisations, which deal with files containing variable length records, the statement is only needed when processing an existing file. It indicates the largest record that the program is capable of handling.

For BD files, the length of each record is determined by the user program, rather than by information held on the file. The symbol form of the statement is normally used so that the program can supply the record length, and thus control the number of bytes actually read or written.

The length must be coded either as an integer between 1 and 32767 inclusive, or as a symbol. When a symbol is used the statement:

```
02 symbol PIC 9(4) COMP
```

is generated within the FD. The program must then place the record length in bytes in this field before executing a file processing statement which requires it.

Because of the different ways in which the RECORD LENGTH statement may be used, it is documented independently as part of the description of each individual file organisation.

1.2.8 The SIZE Statement

The SIZE statement is required only when creating a new file. The size is coded either as an integer between 0 and 999999999 inclusive, or a symbol. When a symbol is used the statement:

```
02 symbol PIC 9(9) COMP
```

is generated within the FD. The program must then place the size to use in this field before opening the file.

The size is the number of bytes to be allocated to the file extent by the OPEN NEW statement. If a value of 0 is specified, or the SIZE statement is omitted, the extent will be allocated the maximum contiguous space available, otherwise an attempt will be made to allocate the exact number of bytes requested and, if this is not available, the program will be terminated in error.

If 0 is used, together with the symbol option, the actual number of bytes allocated will be returned in the generated field.

The SIZE statement with the symbol option can also be used for an existing file. In this case when the OPEN statement completes satisfactorily the actual number of bytes allocated to the extent will be returned in the generated field.

The "SIZE" of a printer, ie the value returned if 0 is used together with the symbol option, is 231-1 bytes, the largest number which can be stored in a PIC 9(9) COMP field.

Note that if you are creating a number of files on the same direct access unit simultaneously, you should generally make at most one size 0 request for the maximum contiguous space available. Furthermore, this maximum size file should be opened last, otherwise it is likely that the System Manager will allocate it all the space on the volume, and there will be none remaining for the other files.

For this reason you should normally avoid making non-specific size requests for new files on direct access units shared by competing multi-user jobs. This problem can be avoided by making the unit a work or spool unit using the $F SWV instruction. This allows a nominal maximum file size to be established for the unit, and requests for the maximum contiguous space will never allocate more than this size.

1.2.9 The OPTION ERROR and ON ERROR Statements

OPTION ERROR must be coded if you wish your program to regain control following an irrecoverable I/O error on the file. If the ERROR option is in force and an irrecoverable I/O error occurs, exception condition 1 will be signalled in response to the affected file processing statement and your exception handling logic will then gain control. (Normally when OPTION ERROR is not coded your job is immediately terminated following an irrecoverable I/O error.) An ON ERROR statement establishes an intercept routine, which will gain control whenever an I/O error occurs, before the System Manager displays the error message.

1.2.10 Organisation-dependent Statements

Some of the following statements may be required in the FD but they are not explained in detail here, either because they only apply to a single file organisation, or because their meaning varies from organisation to organisation. The statements are:

```
OPTION RESET | OPTION IGNORE

BLOCK CONTAINS number CHARACTERS

KEY IS keyname

KEY LENGTH IS keylength
```

The first two are only used in processing relative sequential files. OPTION RESET allows the file to be re-created by WRITE NEXT statements, so that is can be used, over and over again, as a work file. BLOCK CONTAINS provides conventional double buffering or multi buffering to increase the performance of sequential READ NEXT or WRITE NEXT operations.

The KEY and KEY LENGTH statements are mutually exclusive, KEY LENGTH only being coded for files of indexed sequential organisation. Both statements are used to define the key field employed by random access READ and WRITE statements. The OPTION IGNORE statement is only used with indexed sequential files, and causes logically deleted records to be ignored.

1.3 File Processing Statements

Files are created, input and updated by means of file processing statements coded in the procedure divisions of Global Cobol programs. The syntax of the file processing statements is as follows:

```
OPEN type filename [index area]

WRITE NEXT filename FROM area

WRITE filename FROM area

DELETE filename

REWRITE filename FROM area

READ FIRST filename INTO area

READ LAST filename INTO area

READ NEXT filename INTO area

READ PRIOR filename INTO area

READ filename INTO area

READ PHYSICAL filename INTO area

CLOSE filename [TRUNCATE/DELETE]
```

The type coded in the OPEN statement is either the word NEW, OLD or SHARED. It indicates whether the open operation is attempting to create a new file, obtain exclusive access to a file which already exists, or share the processing of an existing file with other users.

The filename, which appears in every statement, labels the file definition upon which the statement operates.

The index area is an ISAM or DMAM option which is included for completeness only, and will not be discussed further in this general introduction.

The area labels a contiguous area of main storage to be output as a record of the file by WRITE NEXT, WRITE or REWRITE, or input by READ, READ NEXT, READ PRIOR, READ FIRST, READ LAST or READ PHYSICAL.

Table 1.3 shows each file processing statement together with the device types for which it can be used. If you mistakenly execute an inappropriate statement your program will be terminated in error if it is not supported by the file organisation you are using: if the organisation supports it, but it is inappropriate for the device (e.g. a READ on a printer) an irrecoverable I/O error will occur.

1.3.1 Exception Conditions

Most of the statements listed in Table 1.3 can suffer a file operation exception if some abnormal circumstance arises which should be handled specially by the program. File operation exceptions are signalled by $$COND being set equal to 2, and the logic introduced by the ON EXCEPTION statement immediately following the affected file processing statement being activated. If the ON EXCEPTION statement is missing, your program will be terminated in error should an exception occur.

The various circumstances which can lead to a file operation exception are known as file conditions, and are referred to by means of a short

title such as "file not found", "end of file" and so on. The possible file conditions are summarised in the rightmost column of Table 1.3 and are explained in more detail in the subsequent discussion of the particular file processing statement affected.

In addition to a possible file condition, a file processing statement may suffer exception condition 1 if an irrecoverable I/O error occurs and OPTION ERROR has been coded in the FD to indicate that the program is prepared to handle such an eventuality. If OPTION ERROR is not specified the program will be immediately terminated on an irrecoverable I/O error, and need contain no logic to process such a condition.

Note that whenever an exception affects any file processing statement apart from CLOSE, the file definition usually remains unchanged, so that fields that would normally be updated by the operation are undisturbed. This means that an OPEN operation which suffers a exception does not actually change the status of the FD to "open". This is so that corrective action can be taken and the operation retried, if necessary.

A CLOSE operation can only suffer an irrecoverable I/O error if OPTION ERROR is coded. If this occurs, although the status of the file may be unpredictable (if the exception was due to an irrecoverable hardware error, for example) the FD itself is returned to closed status just as if the operation had completed successfully.

Because of the likelihood of an exception being signalled, each file processing statement (apart from perhaps REWRITE and CLOSE) is normally followed by an ON EXCEPTION statement to handle the condition. An error intercept routine, established by an ON ERROR statement in the FD, can be used to treat certain I/O errors on OPEN statements as additional file conditions. This is described in section 1.6 below.

1.3.2 The OPEN Statement

An OPEN statement must be executed before any other file processing statement operating upon the file definition. If an attempt is made to read, write or close a file definition which has not been opened your program will be terminated in error. Similarly it is illegal to attempt to open an FD which is already open.

FILE PROCESSING STATEMENT	DEVICE TYPE	POSSIBLE FILE CONDITIONS (exceptions with $$COND = 2)
OPEN NEW	direct access printer	already exists none
OPEN OLD or	direct access	file not found or wrong type*

OPEN SHARED	printer	none
WRITE NEXT	direct access	file space exhausted
	printer	alignment
WRITE	direct access	file boundary violation or file space exhausted
DELETE	direct access	none
REWRITE	direct access	none
READ FIRST	direct access	end of file
READ LAST	direct access	start of file
READ NEXT	direct access	end of file
READ PRIOR	direct access	start of file
READ	direct access	file boundary violation or record not found
READ PHYSICAL	direct access	file boundary violation
CLOSE	direct access	file not open
	printer	file not open

* In these cases the value of system variable $$RES can be used to distinguish between the different causes of the exception:

```
$$RES = "1"   ... wrong type
$$RES = "3"   ... file not found
```

Table 1.3 File Processing Statement Summary

OPEN begins by checking whether the unit-id you specified in your FD's UNIT clause is actually assigned to a physical unit address. If this is not yet the case the operator will be asked to make the assignment. If he cannot do so, he will reply appropriately to the assignment prompt and the System Manager will signal an irrecoverable I/O error.

When unit assignment has completed satisfactorily, volume-id checking takes place providing it is specified by VOLUME clause information in the FD, and the unit involved is a direct access device. The operator will be asked to mount the required volume if it is not already online. If he cannot do so, he may reply appropriately to the mount prompt, and the System Manager will signal an irrecoverable I/O error.

When unit assignment and volume-id checking, if specified, have taken place and all is well, the common part of the open operation is complete. The remaining processing depends on the type of open taking place, ie whether the file is NEW, OLD or SHARED.

A successful OPEN NEW creates a new file with an extent size determined by the FD's size statement. However, the file already exists condition will be signalled, and the open will not take place, if the volume involved is a direct access volume already possessing a file with the same file-id as the one specified in the FD's ASSIGN statement. This file condition prevents you mistakenly creating duplicate file-ids on a direct access volume. Your exception handling logic should determine whether the file is genuinely unwanted and, if it is, delete it using the DELE$ system routine and re-issue the OPEN NEW. (For some situations, the OPDE$ routine might be appropriate to perform such activity with operator control over the deletion.)

If you call an OPEN NEW operation having set the last two bytes of the file-id to LOW-VALUES, the System Manager will create a new file substituting the last two bytes with the file label number, returning the full file name. This is especially useful when creating work files which require a unique filename on the unit.

OPEN OLD is used to obtain exclusive access to a file which already exists. The file not found or wrong type condition will be signalled if a direct access volume does not contain a file with the specified file-id or organisation respectively. These file conditions cannot arise on a printer.

OPEN SHARED is employed in multi-user working when a direct access file may be simultaneously open by a number of competing yet co-operating jobs. An OPEN SHARED statement issued for a printer, which by its very nature cannot be shared, is treated in exactly the same manner as OPEN OLD. The file not found or wrong type condition is signalled in the normal way if a direct access volume does not contain a file with the specified file-id or organisation.

An I/O error will be signalled if OPEN SHARED is issued for a direct access file which is currently open as NEW or OLD. To share a file it is necessary that all its users acquire access to it by means of an OPEN SHARED statement. A program executing an OPEN OLD will suffer an I/O error if the file currently has any other users. Otherwise it will obtain exclusive access to the file during the time it is open, and during this period other users attempting to open the file will suffer I/O errors.

The unshareable nature of printers means that in a multi-user system such devices are allocated in a first come, first served fashion. To allow a number of print jobs to operate at the same time the System Manager provides a spooling facility. Each real printer is allocated to the $SP command program which can be run as a background job. Each copy of $SP is responsible for spooling print files from a specified direct access device to the printer it services. Programs producing print files simply write them to the direct access unit associated with a particular $SP, from which they will be printed in due course. (For more detail see the documentation of $SP in the Operating and Utilities Manuals.)

1.3.3 The WRITE NEXT Statement

The WRITE NEXT statement is provided to allow you to create the records of a file sequentially or, in the case of direct access files only, extend the file with new records. The file space exhausted condition is signalled if the WRITE NEXT statement would, were it

attempted, cause part or all of its record to be written outside the extent.

In the case of a printer the alignment condition is signalled if the operator replies N to the prompt generated as a result of WRITE NEXT outputting a forms control record to control the alignment of special stationery.

1.3.4 The WRITE Statement

The WRITE statement is employed to write a record at random to a direct access file according to a key field you establish in either the file definition or the record area itself. A WRITE statement cannot be used on a printer.

The file boundary violation condition is signalled, for organisations other than indexed sequential, if the key you establish addresses a record wholly or partially outside the used part of the file. For indexed sequential the WRITE statement will cause a new record to be added to the file if one with the indicated key does not exist. In this case the file space exhausted condition occurs if there is insufficient space in the overflow area to hold the new record.

1.3.5 The DELETE Statement

The DELETE statement can only be employed on certain file organisations and only applies to direct access devices. It is used to delete the last record input by means of any READ type statement.

1.3.6 The REWRITE Statement

The REWRITE statement can only be employed on certain file organisations, and only applies to direct access devices. It is used to update the record last input by means of any READ type statement.

1.3.7 The READ FIRST and READ LAST Statements

The READ FIRST statement is used to input the very first record in the file, and will signal end of file if no records are present on the file. Similarly READ LAST is used to input the very last record on the file, and will signal start of file if no records are present on the file. Both READ FIRST and READ LAST may only be used on direct access files, and they are only supported for certain file organisations.

1.3.8 The READ NEXT and READ PRIOR Statements

The READ NEXT and READ PRIOR statements allow you to read the records of a file sequentially. They can be used on direct access devices, but not on printers. READ NEXT returns the next record on the file. Once the last record has been input a subsequent READ NEXT will not cause further information to be retrieved from the file, but will result in the end of file condition being signalled. READ PRIOR returns the previous record on the file. Once the first record has been input a subsequent READ PRIOR will not cause further information to be retrieved from the file, but will result in the start of file condition being signalled. READ PRIOR can only be used with certain file organisations.

1.3.9 The READ Statement

The READ statement is employed to read a record at random from a direct access file according to a key field you establish in either the file definition or record area itself. A READ statement cannot be used on a printer.

The file boundary violation condition is signalled, for most organisations, if the key you establish addresses a record wholly or partially outside the used part of the file. For Indexed Sequential, Data Library and DMAM files, access is controlled by means of a symbolic key and the equivalent file condition is record not found, indicating that there is no record on the file with the key you have specified.

1.3.10 The READ PHYSICAL Statement

The READ PHYSICAL statement is only available for certain file organisations, and only on direct access devices. It is used to read a record at random from a file bypassing some of the normal file handling and going directly to an identified location within the file.

The file boundary violation condition is signalled if the identified location addresses a record which lies wholly or partially outside the used part of the file.

1.3.11 The CLOSE Statement

The CLOSE statement is used to terminate the processing of a file. There are three options: CLOSE, CLOSE...DELETE and CLOSE...TRUNCATE.

CLOSE by itself is always valid, for all device types and organisations. It will complete any outstanding I/O operations and return the file definition to closed status so that it can be used again, should you so wish. The file you have just processed will remain in existence.

CLOSE...DELETE is used to close the file definition and delete the file itself from its direct access volume. The option has no meaning for printers, where it defaults to a normal CLOSE. The CLOSE...DELETE statement should not be issued for a direct access file open by more than one user, since none of the sharing users has the right to delete the common file. An attempt to do so will lead to the offending program being terminated in error.

CLOSE...TRUNCATE closes the file definition and, in the case of a direct access device only, frees any spare space at the end of the extent so that it can be subsequently re-allocated. The TRUNCATE option is ignored when issued for a printer, where the effect of the statement is the same as a normal CLOSE.

Very often when you create a new file you specify a zero size, or omit the FD's SIZE statement, in order that OPEN NEW will allocate it the maximum amount of contiguous space available on a direct access volume. When you have finished outputting the file, providing you do not require to extend it later, you will normally issue a CLOSE...TRUNCATE to release all the unused space for subsequent reallocation. This prevents you having to specify a file size before you begin, and normally leads to the most efficient use of the direct access storage available. (See also the notes on maximum size files in section 1.2.8.) Note that when you CLOSE a file where the size established in the SIZE clause at OPEN time was zero, then the SIZE field is reset to zero to enable your program to easily repeat its processing. Consequently if you wish subsequently to examine the size of the file, you should save a copy of the SIZE field after it is opened, and before it is closed.

The CLOSE...TRUNCATE option can be used on a file originally opened by an OPEN OLD or OPEN SHARED statement. However, if the file is open by more than one user, none of the sharing users has the right to truncate it, and a program attempting to do so will be terminated in error.

If you attempt a CLOSE on an FD which has not previously been opened (during a common close-down routine for example) then the file not open condition will be signalled, and the FD will remain undisturbed.

1.4 Processing Shared Files

Under the System Manager several users may have the same file open simultaneously, by using the OPEN SHARED statement. A shared file can be accessed by READ, READ NEXT, READ PRIOR, READ FIRST, READ LAST or WRITE NEXT statements without any consistency problems arising. If, however, two users are updating the same file using WRITE, DELETE or REWRITE statements and if both attempt to update the same record, one of the updates may be lost. For example, in the simplest case, a program reads, modifies and rewrites a record, but between reading the record and rewriting it the record has been updated by another user. Then the updates performed by the second user will be overwritten by those performed by the first user, and hence will be lost.

The LOCK and UNLOCK statements described below are provided to help you overcome these problems. LOCK enables you to acquire exclusive access to a specified region of a shared file, so that you can make updates without the risk of interference from other users. UNLOCK releases the exclusive access obtained by a previous LOCK.

1.4.1 The LOCK Statement

You use the LOCK statement to obtain exclusive access to a specified region of a shared file. You code:

```
LOCK filename region [WAIT]
```

Here filename identifies the file definition of a file which must have been opened previously, otherwise the program will be terminated in error. If the file was not OPEN SHARED the operation will be ignored. The second parameter is the name of a 4-byte region code used to define the part of the file to which exclusive access is required. Typically, the region code might be:

- The PIC 9(9) COMP record key of the file record that you require to update. The effect of the LOCK statement will then be a record-level lock;

- A PIC X(4) subset of a longer indexed sequential key field. The LOCK statement will then grant exclusive access to records whose keys contain the subset value you have specified.

Providing the region is not locked by another program the statement signals normal completion and grants you exclusive access to it. The region then appears locked as far as other users are concerned.

If the region is locked by another user when your LOCK statement is executed, then, providing the optional WAIT phrase has not been coded the statement signals exception condition 2. If all locks are in use, and the WAIT phrase is omitted, exception condition 1 is signalled.

You can then take appropriate action, such as informing the operator that the account he requires to update is busy, and that he should try again later.

If WAIT is coded and the region is locked by another user, your program is suspended until the region is unlocked. LOCK...WAIT therefore always eventually acquires exclusive access to the region, whereas an ordinary lock can fail with an exception. However, you should take care only to use the WAIT phrase in situations where the updating technique has been designed so that locks are only in force for very short periods, otherwise LOCK...WAIT may cause your program to be suspended for a very long time.

1.4.2 The UNLOCK Statement

You use the UNLOCK statement to relinquish the exclusive access you obtained by a previous LOCK statement. You code:

```
UNLOCK filename region
```

The filename and region must be the same as those specified in a previously successful LOCK statement, and the file definition identified by the filename must still be open, otherwise your program will be terminated in error. If the file was not OPEN SHARED the operation will be ignored. Note also that if a file is locked more than once at the same region it must be unlocked the same number of times to release all the locks.

Normally you should UNLOCK regions as soon as you no longer require exclusive access, in order not to lock other users out unnecessarily. Any locks still in force when a file is closed are automatically unlocked. Therefore, if you use two or more FDs to access the same shared file you should take care that you do not close any one of them during a period when you require exclusive access. There will be no problem if you close all the FDs involved at the same time, once your program is finished with the file.

Note that any locks still in force when your program terminates are automatically unlocked by the System Manager.

1.4.3 Updating and Extending Shared Files

When you need to update existing records on a shared file, or extend it with one or more records, it is essential that you use LOCK and UNLOCK statements to ensure consistency. You should note that the LOCK statement does not, in itself, stop other programs from reading and writing records of the affected region. It simply prevents a LOCK statement for the same file and region succeeding until you have unlocked it. The System Manager therefore has no knowledge of the intrinsic meaning of the region code: it treats it simply as a 4-byte identifier.

The exclusive access mechanism provided by the LOCK statement relies on the same region code convention being adopted by each group of programs which can simultaneously amend the file. Each must determine the region code, LOCK, amend and UNLOCK. Should any program write to the file without doing this, inconsistent updating will result.

It is up to you whether you use the LOCK mechanism to prevent users making enquiries on a region which is being updated. If you want to restrict read access in this way, you must code your programs so that

the region is locked before any file operation takes place on it. More normally, you would only lock immediately before reading the first record to be updated.

The choice of region coding for a particular file updating scheme depends on your application, although if you intend to modify shared files used by other Global software products, you will have to conform to the conventions they use. These are described in the relevant user manuals.

For RS files the usual technique is to use the record number which has to be stored in the PIC 9(9) COMP key field of the FD before a random read takes place. A typical update sequence for FD SALES, key SALESKEY, would then conform to the pattern:

```
LOCK SALES SALESKEY
ON EXCEPTION
DISPLAY "RECORD BUSY, TRY AGAIN LATER"
GO TO etc, etc.
END
READ SALES INTO RECAREA

Display the record, and interrogate the operator to determine the
necessary amendments.
WRITE SALES FROM RECAREA
UNLOCK SALES SALESKEY
```

If you need to extend the file, you use WRITE NEXT statements which do not rely on a record number key. In this case you impose the convention that the region whose code is four bytes of HIGH-VALUES must be acquired before extending the file. Assuming extension to be a rapid process requiring locks of only brief duration, the typical updating sequence might be:

```
LOCK SALES #FFFFFFFF WAIT
WRITE NEXT SALES FROM REC1
WRITE NEXT SALES FROM REC2
UNLOCK SALES #FFFFFFFF
```

When you require to update an IS file, then providing the key is four bytes or less, you can make the region code the same, and thus achieve record level locking. If the key is longer, you can only employ a subset of it to be the region code, and in this case a single LOCK statement will grant exclusive write access to a multi-record region. This can be an advantage. For example, supposing the key is a four-byte customer number followed by a transaction number.

If there are operations which involve updating or inserting a number of transactions for the same customer, then by using the customer number as the region code, you ensure that a single LOCK statement grants exclusive access to all the records relevant to a particular customer.

When you require to lock a single record of an IS file with a key longer than 4 bytes there are several strategies available to you. For instance, you can select four bytes of the key which are known to differ between records, or you can employ some type of hashing technique to convert the long key into a PIC 9(9) COMP number to use for the region code. Providing you choose the encoding technique

carefully it will be extremely unlikely that you will accidentally lock a second record that another program wishes to update.

Sometimes you wish to implement a scheme in which, for a short period, only one operator is allowed to update a file, but any number are allowed to read it. In this case you should treat the entire file as a single region, named "FILE" in the example which follows. When an operator attempts to enter update mode the program in control executes a sequence such as:

```
LOCK CUSTOMER "FILE"
ON EXCEPTION
DISPLAY "ANOTHER OPERATOR IS UPDATING THE CUSTOMER FILE"
etc, etc.
END
```

When the updates are complete, exclusive control of the process is relinquished by means of:

```
UNLOCK CUSTOMER "FILE"
```

The programs used by enquiries only READ the file and do not LOCK it.

1.4.4 Programming Notes

It is vital that all groups of programs which can update a particular file at the same time use the same region convention. Thus, for example, errors would result if you tried to mix the "FILE" locking technique with programs which updated regions identified by a customer number.

If you need to update several regions at the same time you will have a number of LOCK statements outstanding at once. In this case you must be careful to avoid the "deadly embrace" situation which can come about if program 1 attempts to lock regions A and B, in that order, when program 2 is trying to lock B and then A. There is a danger of an impasse: program 1 may acquire region A, and program 2 region B, and, if LOCK...WAIT has been used, both will be suspended indefinitely.

The deadly embrace can be avoided if you impose a convention that locks must always be acquired in a particular order - e.g. lower region codes before higher ones, and all locks on file A before any locks on file B. Another technique, which does not rely on ordering, is to avoid use of the WAIT phrase and insist that whenever a program fails to obtain a desired lock it unlocks any other regions it has previously locked, suspends itself temporarily, and then tries again to obtain all the locks required.

Internal table size limitations restrict the maximum number of locks outstanding at any one time. If you attempt to lock a region, and all locks are currently in use, your request will not be successful and the LOCK statement will be treated just as though the region were in use by another program: an exception will be signalled, or LOCK...WAIT will be suspended until an UNLOCK issued by another user cancels one of the existing locks.

A typical System Manager system is configured with 100 locks available, but this figure may be increased by using Global Configurator.

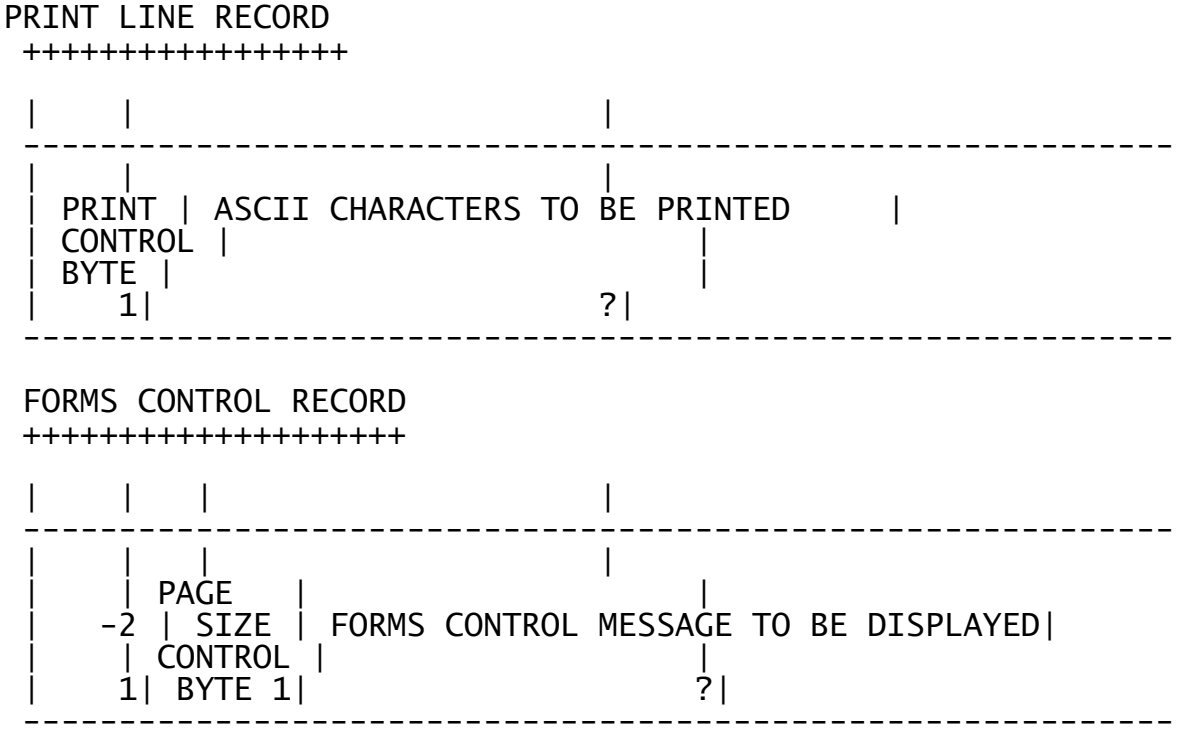
As well as the exclusive locks documented here there are also shared locks, used in master/servant record updating, which are documented in section 9.18.

1.5 Print Files

Information to be printed must be written to a print file using the relative sequential organisation. Whether the data is actually printed immediately or stored on a direct access device or magnetic tape for subsequent spooling depends on the unit address assigned to the file when your program comes to be run.

1.5.1 Record Format

A print file is made up of fixed length print line records or forms control records, of the format pictured below:



The print control byte is a PIC S9(2) COMP field normally containing a value between -1 and 31. The value -1 causes a skip to a new page. A value of 0 means that the ASCII characters in the rest of the record will be output on the current line, overprinting the line previously printed. A value, p, between 1 and 31 signifies that the paper is to be advanced by p lines before printing the information in the record. Note that some printers cannot support overprinting, and for these printers overprint lines will be printed on the next line instead.

If the size of the print line record indicates that there are more ASCII characters to be printed than there are character positions on the printer, the rightmost characters of each record will be lost. If there are more character positions available than characters to be printed, the rightmost character positions will remain blank.

A print control byte with value -2 is used to denote a forms control record. Print control bytes with values of -3, -4 , -5, -8, -9, -11 and -12 have special meanings, and are covered at the end of this section. Values not in the range -14 to 31 cause a single line advance

(i.e. are treated as though a 1 had appeared). This is to prevent unwanted erroneous paper throws during program testing.

In the forms control record the page size control byte is a PIC S9(2) COMP field containing either the values -1, or 0, or the number of lines per page of special stationery, which must be no greater than 99. The value -1 is used to indicate a stationery alignment message to which the operator may reply N to cause an alignment pattern page to be reprinted. The non-negative values indicate a stationery mount message, which requests special stationery to be loaded, and specify the number of lines per page of the paper in question. A zero value indicates that the size is standard and that the normal installation page size (contained in the system variable $$PAGE, normally 66 lines per page) is to be used.

The forms control message is used to inform the operator of the stationery or alignment requirements. If the record length of the print file is less than 52 bytes, the forms control message length will be correspondingly reduced. If the record length is 52 bytes or more, the message length is exactly 50 bytes with rightmost blanks if necessary.

Under System Manager V6.2 and later the first two characters of a stationery mount message can have a special meaning. If the first character is an upper case letter (A-Z) and the second (separator) character is a slash (/) or equals sign (=), then the letter is taken to identify the format letter which will be used in printing the report. The setting up of format letters is described under the $CUS documentation in the Operating Manual. The format letter will condition the response of the printer to requests for Bold characters, and other emphases. If no format letter is specified then the default letter (set up using $CUS) will be used. The slash separator is used to indicate that there is no alignment pattern following this record in the print file (this case is assumed when no format letter is specified in V6.2 and later systems), and the equals separator indicates that there is an alignment pattern following. The System Manager uses this information to suppress alignment pattern printing when no stationery mount is in fact required (see later sections on pagination and special stationery for more details).

There are options in the spooler and $CUS which can directly affect the printing of special stationery. These are fully documented under $CUS and $SP.

1.5.2 The File Definition

The file definition for a print file forms a subset of the full relative sequential definition described in Chapter 2, and should be coded as follows:

```
FD filename ORGANISATION RELATIVE-SEQUENTIAL
[ASSIGN TO UNIT unit-id FILE file-id [VOLUME volume-id]]
RECORD LENGTH IS length
[SIZE IS size]
[BLOCK CONTAINS number CHARACTERS]
[OPTION ERROR [RESET]] | [OPTION RESET [ERROR]]
[ON ERROR intercept]
```

The length is the number of ASCII characters to be printed per line, plus one byte (to leave space for the print control byte). It must be

between 2 and 255 inclusive. The value normally coded is 133, for printing full width 132 character reports.

Normally the SIZE statement should be omitted, and in this case a file assigned to a printer will be allocated a very large extent so that the file space exhausted condition can never occur. (If the file can in some cases be assigned to a direct access device you may, of course, need to make a specific request.)

OPTION ERROR should be coded only if you require your program to regain control (with exception condition 1) following an irrecoverable I/O error. Similarly the ON ERROR statement should only be coded if you wish to process certain I/O error conditions specially.

The VOLUME clause, BLOCK CONTAINS statement and OPTION RESET are not used when the file is written to a real printer, but may be present if in some circumstances it is to be assigned to a direct access device.

If the print file is assigned to a spool unit then a number of special effects may be produced by placing particular characters into the file-id:

- If the seventh character of the file-id is ";" and the eighth a digit, this digit indicates the priority to be given to the file when it is printed by the spooler (1 = high, 9 = low). If the eighth character is "H" this will instead cause the file to be Held by the spooler, requiring an operator to release it before it will be printed;

- If the sixth character of the file-id is ";" and the seventh and eighth characters are a valid number (such as " 1" or "34"), then the number indicates how many copies of the file are to be printed (from 1 to 99). This can be used to save time when printing multiple copies of a report via the spooler;

- If the fifth character of the file-id is ";" then the sixth and seventh characters indicate a number of copies, and the eighth character a priority or held status, in a similar way.

For example, a file named SAPRIN;3 will be given priority 3, a file named SAPRI;22 will have 22 copies printed, and a file named SAPR; 35 will have 3 copies printed at priority 5.

If the file name is not in this format, the file is given the default priority, usually 5, and one copy will be printed.

1.5.3 Opening the Print File

To open the print file it is recommended you code:

```
OPEN NEW filename
```

The file already exists condition may be signalled if the print file is assigned to a direct access unit which is not a spool unit. It cannot occur for a real printer or spool unit. An I/O error will be signalled if the file is assigned to a real printer which is already in use.

(An OPEN OLD or OPEN SHARED issued for a print file will succeed if the file is assigned to a real printer not already in use, signal file

not found if assigned to a spool unit, and succeed or signal file not found as appropriate if assigned to a direct access device which is not a spool unit.)

1.5.4 Writing Print Lines

Print lines are output using the WRITE NEXT (not WRITE) statement to transfer a print line record from main storage to the device:

```
WRITE NEXT filename FROM area
```

The first such record to be output should contain a print control byte of -1 so that the report begins at the top of a new page.

If the FD makes a specific request for file space (as may be the case if the file is sometimes written to a direct access device) it is possible that the WRITE NEXT operation may suffer the file space exhausted condition. If you require to handle this eventuality code an ON EXCEPTION statement following the WRITE NEXT: if you do not your program will be terminated in error should the condition arise.

1.5.5 Forms Control Prompts

The WRITE NEXT statement can also be used to output a forms control record. When the print file is assigned to a direct access device the record is simply stored normally. However, when it is directed to a real printer the record is intercepted and is not actually printed but is used to modify the behaviour of the printer.

A stationery mount request (page size 0 or positive) causes a temporary adjustment to the current page size (as explained below) and to displays a forms control prompt:

```
PLEASE MOUNT forms-control format-extension ON UNIT address:
```

The forms-control element comes from your forms control record, and the format-extension is provided by the format letter you chose (or the default if you did not choose one explicitly), for example:

```
PLEASE MOUNT A4 Letterhead with cartridge A ON UNIT 502:
```

The paper is aligned at the top of the page and then your program is stopped, awaiting the operator's reply. If he or she responds with Y, <CR> (or any single character apart from N) the WRITE NEXT statement completes normally and the program continues. A reply of N indicates that the printer or paper is unavailable for some reason and causes the System Manager to signal an irrecoverable I/O error condition. Your program will only regain control, with exception condition 1, if OPTION ERROR has been coded in the FD.

On V6.2 and later systems, if the format of the stationery is unchanged from the previous stationery then the mount message will not appear.

If the forms control message indicates an alignment check (page length of -1) then the System Manager displays a message of the form:

```
forms-control ON UNIT address:
```

which might appear as

```
IS ALIGNMENT CORRECT ON UNIT 502:
```

Again a reply of Y or <CR> causes the program to continue normally, but a reply of N indicates that the paper is not yet positioned satisfactorily. It causes an alignment file condition, exception condition 2 (rather than an irrecoverable I/O error). Your program must honour this by reprinting the alignment pattern and rewriting the forms control record so that the process of paper positioning can be repeated, over and over again if necessary, until the stationery is eventually aligned satisfactorily.

The use of forms control records, and the reaction of the operating system to them, is described in more detail in the sections below on pagination and the handling of special stationery.

1.5.6 Closing the Print File

A CLOSE statement of the form:

```
CLOSE filename [DELETE | TRUNCATE]
```

must be executed to close the print file once the report is complete. The DELETE and TRUNCATE options are ignored when the file is written to a real printer.

When the device is a printer, CLOSE causes a skip to a new page, to make it easier to remove a completed report. However, if the next record written to the printer itself requests a page advance, The System Manager will remember that the previous CLOSE has actually positioned the paper correctly, and the stationery will not be moved. This prevents unnecessary blank pages appearing between reports.

1.5.7 Pagination

When a print file is opened on a real printer the System Manager assumes that the stationery and page size to be used are the default values defined in the printer control file for the device in question (if there is no printer control file, on a pre-V6.2 system for example, then the System Manager assumes 'standard' stationery and page size). Usually the number of lines per normal page is the value given in the system variable $$PAGE, but it is possible for a configuration to support one or more special printers whose normal page size is different from the installation's standard $$PAGE size.

It is important to realise that the pagination of a report is entirely under the control of the application program. Every time a print line record with a -1 control byte is output, a new page begins. The System Manager will not automatically skip to a new page if you write more lines than the page size, and in this case you are liable to print over the stationery perforations. The first print line record of each print file should start with a -1 control byte so that the line it contains begins on a new page, and from then on your program must ensure that only the correct number of lines are printed on each page.

When you need to use special stationery you may temporarily override the printer's normal page size by specifying the value you require in the page size control byte of a forms control record written to handle the loading of the paper, as explained in 1.5.8. Part of the processing of such a record involves the temporary updating of an internal current page size field associated with the printer.

The current page size field is used when the System Manager is handling stationery with a special page size so that the appropriate number of line feeds can be output to honour a request to advance to new page. Usually, when normal paper is loaded, the printer hardware operates a form feed mechanism which is able to position at the top of a new page somewhat faster than can be achieved by multiple line feeds. However, this feature is not present on all printers.

When the System Manager needs to use multiple line feeds for page advancement it maintains an internal counter for each printer to remember how much of the current page has been used. To set the counter initially the System Manager assumes that the printer is positioned at the top of a new page at the start of a session, and whenever the operator replies to a forms control prompt.

System Manager V6.2, and later, keeps track of the last stationery (and format letter) used on a particular printer provided that a print control file (even an empty one) has been set up for the printer. Therefore if the stationery has not changed between two reports the mount message on the second report will be suppressed. A stationery change is signalled by either a change in the stationery mount message in the forms control record or by a change in the mount message extension in the printer control file. If these two messages are the same as that of the previous file printed then a mount message will not be displayed even if the format letter has changed. However, for V8.1 System Manager and later you can force a mount message to be displayed by using a print control byte of -12 rather than -2 for the forms control line.

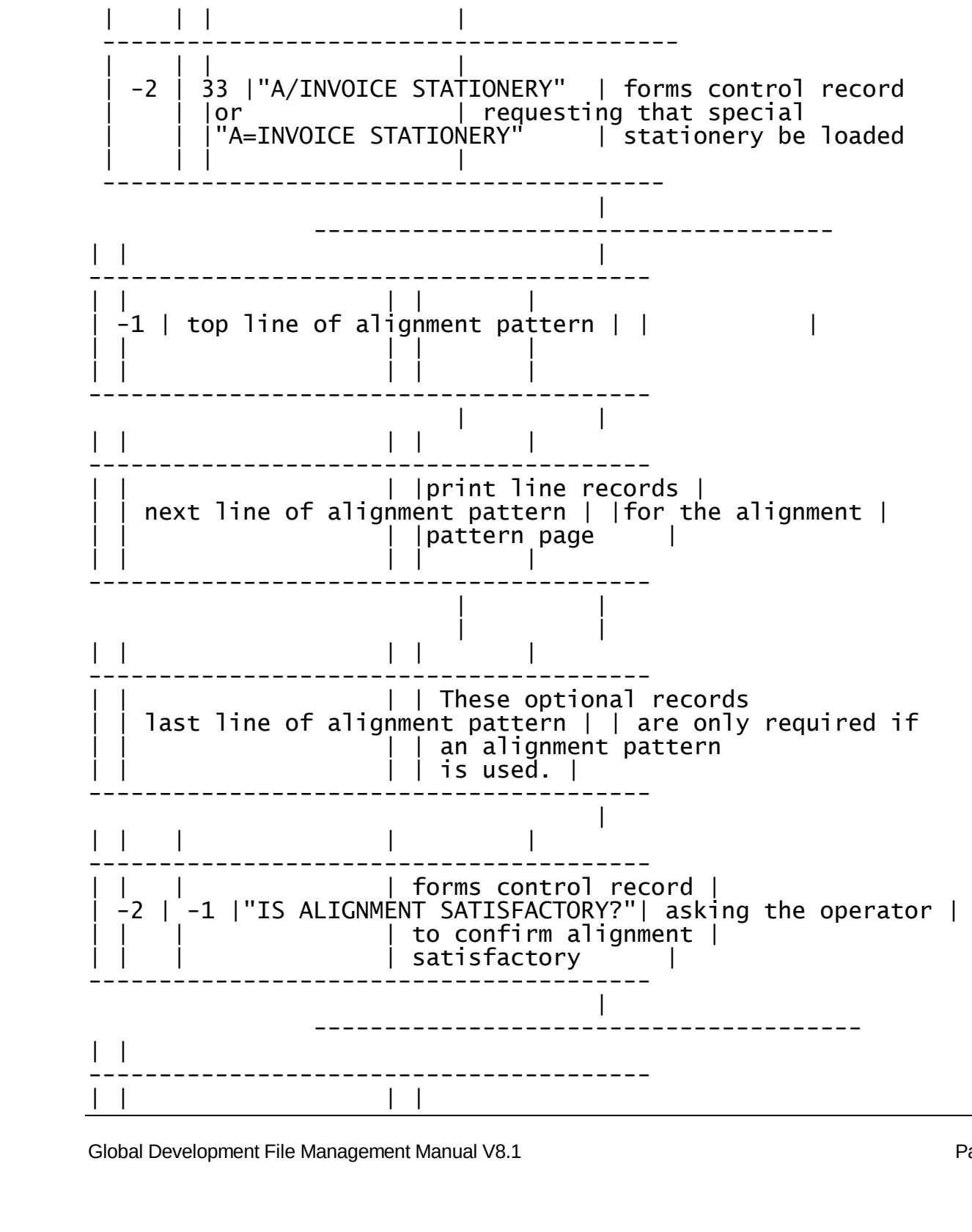
For V8.1 and later systems it is possible to set a printer so that no mount messages are produced using $CUS. This option in $CUS will override the force mount message print line and will also mean that any stationery changes will not be recognised and the printer will continue printing. Therefore this option must be used with great care. (See $CUS documentation).

When a report is printed using standard stationery (that is with no special stationery record or with a special stationery record that does not contain a format letter and separator), the System Manager substitutes the values established in the printer control file to create its own stationery mount message. If the spooler is running, and that has a default printer letter set up, then this letter will be taken in preference to the default letter. The separator for these default stationery records is taken as "/".

If a report indicates to System Manager that it has an alignment pattern, by use of the '=' separator in the forms control message, then System Manager also suppresses printing of the alignment pattern when the stationery mount message is suppressed. When System Manager suppresses an alignment pattern it ignores all print lines following the initial forms control record until the alignment check forms control record is encountered. Care must be taken to ensure that there is an alignment check record printed as the absence of one will mean that none of the file will be printed. If this would cause problems with your report printing then you must not use this feature. If you use a slash separator rather than equals, and you do have an alignment pattern then System Manager will then print the alignment pattern always, but will still suppress the alignment check prompt when it arises).

If you are using stationery with pre-V6.2 stationery mount messages, the default format type will be taken and the default separator will be set to "/". For V8.1 and later systems there is an option in the spooler which allows the default separator for pre-V6.2 stationery mount messages to be "=" instead or "/". Care should be taken when using this option as the absence of an alignment check record may mean that none of the file will be printed. (See spooler documentation)

If you are using special stationery you should always set up a printer control file using $CUS even if there are no special sequences to set up. If you do not, then the mount stationery messages will always appear on the screen.



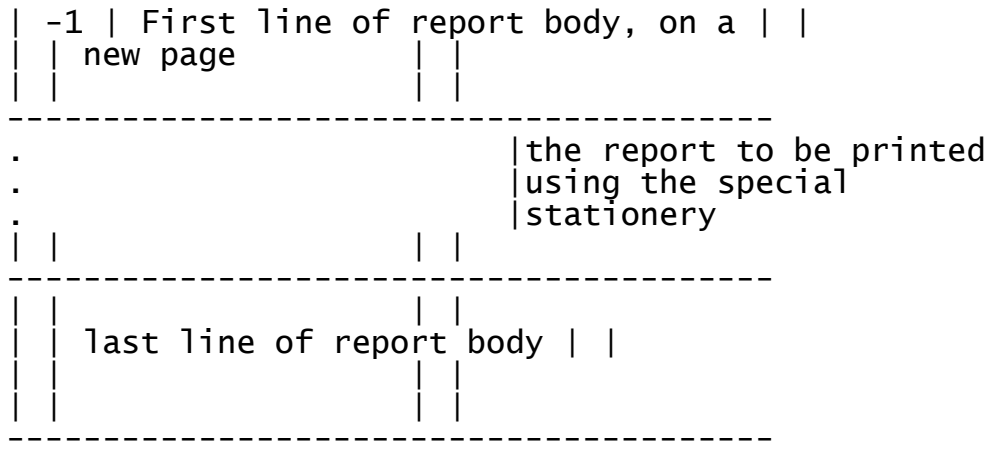


Figure 1.5.8 - A Print File using Special Stationery

1.5.8 Print Files using Special Stationery

If a program requires to use special stationery it should write the whole report involved to a single print file beginning with a forms control record containing the page size of the special paper and a message requesting the operator to load it.

For example, if the record is described in working storage by:

```
01    LOAD                 * LOAD INVOICES
      03 FILLER     PIC S9(2) COMP
                VALUE -2
      03 LO-SIZE    PIC S9(2) COMP
                VALUE 33 * 33 LINES/PAGE
      03 LO-FORM    PIC X      * FORMAT LETTER
                VALUE "A"
      03 LO-SLAS    PIC X      * "/"
                VALUE "/"
      03 LO-MSG     PIC X(129)
                VALUE "INVOICE STATIONERY"
```

then, if the print file is output to a real printer at unit address 500, the current page size will be set to 33, format letter A will be selected and the following prompt will be displayed at the console (assuming format A has the extension "WITH CARTRIDGE A" set up in its definition):

```
     PLEASE MOUNT INVOICE STATIONERY WITH CARTRIDGE A ON UNIT 500:
```

The operator will load the required stationery and then key Y, <CR> (or any single character apart from N) to continue. If he or she replies N, The System Manager will signal an irrecoverable I/O error, indicating the special paper is not available. You may wish to replace the "/" with "=" (see 1.5.7 - pagination).

It is usual (although not obligatory) to print an initial alignment pattern page to check that the fields of the report which follows will be registered correctly. The pattern, normally made up of groups of Xs or asterisks, can be repeated over and over again if necessary, until the paper is positioned correctly. A print file starting with an alignment pattern should be structured as shown in Figure 1.5.8. Providing this is done the file may initially be written to direct access storage knowing that the System Manager $PRINT or $SP utilities

will handle the forms control and alignment correctly when the file eventually comes to be printed.

The initial forms control record requesting the special stationery to be loaded should be followed by the print line records which make up the alignment pattern page itself. Then comes a second forms control record used to prompt the operator to confirm that the alignment is satisfactory. For example, if the record is described in working storage by:

```
01 ALIGN        * ALIGN INVOICES
      03 FILLER     PIC 9(2) COMP
                VALUE -2
      03 AL-SIZE    PIC S9(2) COMP
                VALUE -1 * SPECIAL ALIGNMENT
                      * PROMPT
      03 AL-MSG     PIC X(131)
                VALUE "IS ALIGNMENT SATISFACTORY?"
```

then, when it is written to unit 500, the following prompt appears:

```
     IS ALIGNMENT SATISFACTORY? ON UNIT 500:
```

If the operator replies Y, <CR> (or any single character apart from N) the program will continue by outputting the report which follows. However, if he responds N, then since a special forms control record containing a page size of -1 has been used, the System Manager will signal the alignment file condition. The program will then reprint the alignment pattern page and output the second forms control record once more. This processing will be repeated over and over again if necessary, until the operator responds positively to the prompt, indicating that the paper has been positioned correctly.

Any print file which causes special stationery to be mounted should be terminated with a forms control record requesting that standard stationery be reloaded if the program is to run on pre-V6.2 systems. For compatibility with pre-V6.2 systems - System Manager V6.2 and later will suppress such forms control records and perform its own handling of the stationery mounts required via the printer control file. However, if the program is not to be run on pre-V6.2 System Manager, this last mount standard stationery should be omitted. If more than one type of special paper is required by a program, separate print files should be written for each type of stationery, to ensure that the files will be handled correctly if written to a spool unit.

Mount stationery records must not appear in the middle of print files on V6.2 or later versions of System Manager. However, it may be that some pre-V6.2 applications may contain forms control messages in the middle of files which they expect to be honoured. There is a special $CUS option on V8.1 or later systems for a particular printer which allows mount messages in the middle of files to be honoured but it must be noted that any change in format letter within forms control lines in the middle of files will not be done. (see $CUS documentation)

1.5.9 Special Print Control Values

Values of -3, -4, -5, -8 and -9 in the print control byte have the following special meanings:

- A value of -3 indicates that the record contains a start sequence. The record is sent to the printer but no line-feed takes place, and no attempt is made to translate any of the characters. Any data set up by a previous record with a print control byte of -4 is forgotten;

- A value of -4 indicates a special set up record. This contains data about character translations which will be remembered by the executive for this printer until it is superseded, or removed by a start sequence record. The set up record should always be 133 characters long in the following format:

```
01 SET-UP-RECORD
          02 SEPCB  PIC 9 COMP     * print control byte
                    VALUE -4
          02 SETR   OCCURS 16      * up to 16
                                   * translations
          03 SETCH  PIC X          * char. to translate
          03 SETSQ  PIC X(4)       * translation
                                   * sequence
     *
          02 SEGR                  * graphics data
          03 SEGONL       PIC 9 COMP     * length of graphics
                                   * on
          03 SEGON  PIC X(5)       * graphics on
                                   * sequence
          03 SEGOFL       PIC 9 COMP     * length of graphics
                                   * off
          03 SEGOF  PIC X(5)       * graphics off
                                   * sequence
          03 SEGCH OCCURS 16  PIC X     * line and box
                                   * characters
     *
          02 SEBO                  * bold data
          03 SEBONL       PIC 9 COMP     * length of bold on
          03 SEBON  PIC X(5)       * bold on sequence
          03 SEBOFL       PIC 9 COMP     * length of bold off
          03 SEBOF  PIC X(5)       * bold off sequence
     *
          02 FILLER       PIC X(12)      * RESERVED for
               VALUE LOW-VALUES    * expansion
```

  The character translations replace the character indicated in SETCH by up to four characters from SETSQ - characters are copied from SETSQ until a #00 (low-values) character is encountered. Any translation with spaces in SETCH is ignored. The graphics information provides a list of characters to replace the standard Global graphic characters (#80 to #8F) along with optional graphics on and off sequences if an alternate character set selection is required. The bold information specifies a bold on and bold off sequence which will replace the characters #90 and #91 respectively if they are encountered in a print line (see 1.5.11 for more details);

  Note: use of records with -3 or -4 print control bytes is not recommended under System Manager V6.2 and later systems. All handling performed by such records (and a great deal more) is performed by the V6.2 printer control file handling, and if records with print control bytes of -3 and -4 are encountered in

a file they will cause the V6.2 printer control file handling to be temporarily disabled.

The System Manager will set the control file to a special undefined state, which will cause the next print file to request a stationery mount regardless of the forms control information used.

- A value of -5 indicates a line which is to be sent to the printer without appending a carriage return. This can be useful either when sending a long programming sequence to a printer, where you do not wish carriage return characters to become embedded into the sequence, or if you need to send a single line to the printer which is wider than the internal System Manager buffer (normally because you have embedded special escape sequences into the line to produce special characters or emphases).

  Bear in mind that a record with a control byte of -5 does not cause any line advance before it is printed, so you would normally need to print a blank line with the correct line advance in its print control byte first.

- Values of -8 and -9 behave as -5 except that for records with print control bytes of -8 no printer translation is applied and for records with print control bytes of -9 no translation is applied to the record and any trailing spaces in the record will be ignored.

These special print control bytes provide facilities which are used by the System Manager itself, and should not normally be needed by programs unless they are performing very complex printer handling (as for example GLOBAL Writer does).

Note that the information contained in a set up record (print control byte of -4) is only sent to the printer when required by other print records. The actual set up record is intercepted by the executive and held in an internal table until it is required.

1.5.10 Opening an Old File on a Spool Unit

Normally, if you attempt an OPEN on a file residing on a spool unit, the OPEN will fail with the file not found condition. The most obvious reason for this is that the System Manager renames files opened on a spool unit with a file-id which consists of a three digit sequence number, the originating operator-id, and the partition number. However even if you were to construct the correct file-id, the OPEN would still fail as there is special processing within the System Manager to prevent re-use of files on the spool unit and avoid interference with the spooler.

If for some reason you need to open a file on the spool unit, you need to construct the appropriate file-id, and then set a special flag within the FD to indicate to the System Manager that it is to allow the OPEN to proceed. You must redefine the FD as follows:

```
01   FILLER REDEFINES filename
  03 FILLER         PIC X(4)
  03 FDERF          PIC 9 COMP
```

Before you issue the OPEN, you must set FDERF to the value -1. Note that this will also cause the FD to be treated as if you had specified

OPTION ERROR, so you will need to be prepared to handle exceptions returned from irrecoverable I/O errors in your program.

1.5.11 Using different emphases and typefaces in printing

Under System Manager V6.2 and later systems there are a number of special characters which may be embedded into a print file to enable the uses of various types of emphasis. An emphasis will usually be a different style of character (italics or bold), or perhaps even a completely different typeface.

There are eight combinations of characters which can be used to enable and disable the various emphases set up in the printer control file for the format letter you are using. These are:

#90 and #91 For Bold on and off, taking up a space in the print line (and hence compatible with thepre-V6.2 usage).

#92 and #93 For Underline on and off, again taking up a space in the print line.

#94 and #95 For Bold on and off which does not take a space in the print line.

#96 and #97 For Underline on and off which does not take a space in the print line.

#98 and #99 For Italics on and off which does not take a space in the print line.

#9A and #9B For Heading on and off which does not take a space in the print line.

#9C and #9D For Superscript on and off which does not take a space in the print line.

#9E and #9F For Subscript on and off which does not take a space in the print line.

The actual emphasis used (Bold, Underline, Italics, Heading, Superscript or Subscript) is only so named by default. You may establish any convention you wish as to the use of the six emphasis sequences, provided you set up your printer control files in an appropriate way, but these are the default uses made of the sequences by System Manager and related GLOBAL products (such as Writer).

The two emphasis pairs which take up a space in the print line are provided so that they may easily be inserted into existing reports without affecting line layout. The non-space taking emphases will more commonly be used by new programs (as they are by Writer).

1.5.12 Printer Hopper and Line Spacing

Under System Manager V8.1 or later it is possible to select a hopper or to change a particular line spacing as defined by the current format letter in the printer control file. This is done by writing the special hopper or line spacing selection line to the print file.

The format of this selection line is as follows follows:

```
      01 HP
       02 HPCB   PIC 9(2) COMP * PRINT CONTROL BYTE
                 VALUE -11
       02 HPHPSP      PIC 9(2) COMP * SELECTION TYPE
                           * 1 - HOPPER 1
                           * 2 - HOPPER 2
                           * 3 - 0 LINE SPACING
                           * 4 - HALF LINE SPACING
                           * 5 - 1 LINE SPACING
                           * 6 - 1 AND A HALF LINE
                           *     SPACING
                           * 7 - DOUBLE LINE SPACING
```

The lines following the selection line will then be printed according to the selection type.

$$RES	ERROR DESCRIPTION	DEVICES	FILE	NOTES
		PROCESSING		
		STATEMENTS		
A	Invalid Access Option	D	OPEN	
B	Error Purging Buffer	D	not OPEN	1
C	Subvolume not allocated	D	OPEN	
D	Invalid Unit Number	D P	OPEN	
E	Out of Paper	P	WRITE NEXT	
F	File in Use	D P	OPEN	
G	Ribbon Out	P	WRITE NEXT	
H	Hardware Error	D P	any	2
I	Interface Error	D P	any	4
J	Hardware Write Protect	D	WRITE (NEXT)	
K	Wrong Record Length or LAN Buffer Error	P	READ NEXT	
L	Network Error	D P	any	3
M	Mount Error or Stationery Not Available	D P	OPEN WRITE NEXT	
N	Operation Sequence Error	D P	any	
O	Invalid Operation	D P	any	
P	Write Protected or Password Protected	D D	WRITE (NEXT) OPEN	
Q	Computer Not Available	D	any	5
R	Read Error	D	any	2
S	Insufficient Space	D	OPEN NEW	
T	Too Many Open Files	D	OPEN	
U	Not Ready	D P	any	
V	Incorrect Volume Format	D	OPEN	
W	Write Error	D P	most	
X	Directory Full	D	OPEN NEW	
Y	Invalid Directory	D	OPEN NEW	
Z	Internal Error	D P	any	4

DEVICES: D = Direct Access P = Printer

NOTES

1. Error B indicates that the previous WRITE NEXT statement suffered an I/O error, and has not been correctly completed. This occurs because records output using WRITE NEXT are normally buffered, rather than written immediately, for performance reasons. If the failing operation is retried it may succeed, but the previous WRITE NEXT will have been partially or totally lost.

2. These errors may occur on an OPEN if the wrong format volume has been mounted, for example double-density instead of single-density diskette.

3. Remote units only ($REMOTE or network).

4. These errors normally indicate that the program or FD has been corrupted due to a programming error. The job should be terminated immediately, rather than attempting to recover.

5. This error will only occur when accessing files on another computer on a System Manager/LAN network.

Table 1.6.1 - I/O Error Result Codes

1.6 Special I/O Error Handling

Normally an irrecoverable I/O error on a file simply terminates execution of the program. This section describes how advanced applications can handle certain types of I/O error themselves, either by attempting to recover from the condition or by performing additional processing to ensure that the system is not left in an inconsistent state.

1.6.1 I/O Error Result Codes

The result code, system variable $$RES PIC X, is set to a value in the range "A" to "Z" whenever an I/O (or similar) error occurs on a file. Table 1.6.1 lists these result codes and their meanings. Two extra conditions are treated like I/O errors and can occur at OPEN time: the result code is set to "M" if the operator replies N to a mount prompt or stationery load request; and the code is set to "P" if he or she fails to satisfy a password prompt.

Note that the result code is also set to values in the range "1" to "6" when a file condition occurs, or when exception condition 2 is signalled by certain system routines, as shown in the table below. The code can therefore be analysed in the exception handling logic following I/O operations and certain system routine calls to determine the precise cause of an error.

$$RES	MEANING	STATEMENTS/ROUTINES
"1"	Wrong type	OPEN OLD, OPEN SHARED, CONV$, PROG$

"2"	Key out of sequence	CONV$	
"3"	File not found	OPEN OLD, OPEN SHARED, CONV$, COPY$, CATA$, PROG$, RENA$	
"4"	File already exists	OPEN NEW, RENA$	
"5"	File capacity exceeded Spool unit (cannot list)	WRITE NEXT, CONV$, COPY$ OPEN$	
"6"	Alignment No System Manager directory	WRITE NEXT OPEN$	

Table 1.6.1A - File Condition Result Codes

1.6.2 The OPTION ERROR Statement

The OPTION ERROR statement may be included in any working storage file definition. If OPTION ERROR is in force and an irrecover able I/O error occurs, exception condition 1 will be signalled in response to the affected file processing statement and your exception handling logic will gain control. (Normally, when OPTION ERROR is not coded, your job is immediately terminated following an irrecoverable I/O error.) The result code, $$RES, can be tested to determine the type of error that occurred. You would normally use OPTION ERROR in programs that take special recovery action when I/O errors are detected, for example to use a backup copy of the record causing the I/O error.

When OPTION ERROR is specified the exception handling logic must distinguish between an I/O error and the normal file exception, for example by coding:

```
READ NEXT MASTER INTO PAYREC
ON EXCEPTION
     GO TO DEPENDING ON $$COND
          TO ERROR        * EXCEPTION 1
          TO ENDFILE           * EXCEPTION 2
END
```

1.6.3 The ON ERROR Statement

The ON ERROR statement may be included in any file definition in order to establish an error intercept routine for the file. It is coded:

```
ON ERROR intercept
```

When the name of the routine is known before the program executes you code intercept as the section name in quotes, for example:

```
ON ERROR "F1-ERR"
```

Normally this will identify a section within the same compilation, but it is possible for the section to reside in some other module participating in the linkage edit, providing its name is defined as a global symbol by using the GLOBAL statement.

If the address of the routine is to be determined at run-time then code intercept as a symbol. This causes the statement:

```
     02 symbol PIC PTR
```

to be generated within the FD. The program using the file is responsible for initialising this field to address the start of the intercept routine. If it is not initialised, or if it is set to LOW-VALUES, then there is no intercept routine associated with the file.

1.6.4 Intercept Routines

An intercept routine, established using the ON ERROR statement in an FD, is executed whenever an I/O error occurs on the associated file, before any error messages are displayed by the System Manager. It must therefore be resident whenever the file is accessed.

An intercept routine can test the result code, $$RES, to determine the type of error, as described in section 1.6.1. The routine should then either return to the System Manager normally, or signal one of two possible exception conditions, as shown in the following table:

STATEMENT	EFFECT
EXIT	the System Manager displays its normal error message and retry prompt, as if there were no intercept routine. routine.
EXIT WITH 1	The System Manager signals an irrecoverable I/O error immediately, without displaying any error message or prompt. If OPTION ERROR is in force then the controlling file processing statement will be suppressed with exception 1; otherwise the job will be terminated immediately.
EXIT WITH 2	Provided the controlling file processing statement is an OPEN, the System Manager suppresses it with exception 2, to signal a file condition, without displaying any error message. Otherwise, if the statement was not an OPEN, the usual error message and retry prompt is displayed.

Table 1.6.4 - Exit from an Intercept Routine

The EXIT WITH 1 statement is normally used in conjunction with OPTION ERROR to suppress unwanted error messages. For example, you may wish to process a file ignoring any records which cause I/O errors when accessed.

The EXIT WITH 2 statement is used to provide special handling of I/O errors occurring at open time. Figure 1.6.4 shows an example program which intercepts error "F", file already in use, and rather than the program being terminated allows the operator to specify a different file to be processed. Similarly a program might intercept error "D", invalid unit address, and allow the operator to input the correct unit address. Note that the exception handling logic following the OPEN statement should test for an alphabetic value of the result code, $$RES, to distinguish the special exception generated by the intercept routine from the normal file already exists, file not found and wrong type exceptions.

An intercept routine must not start with an ENTRY statement, and may only contain the following types of statement:

- arithmetic statements;
- conditional and iterative statements;
- MOVE, EXIT and GO TO statements.
- DISPLAY and DISPLAY...LINE statements.

```
      .
      .
      FD F1 ORGANISATION RELATIVE-SEQUENTIAL
      ASSIGN TO UNIT "DSK" FILE F1-NAME
      ON ERROR "F1-ERR"
      .
      .
      .
      PROCEDURE DIVISION
 AA100.
      DISPLAY "INPUT FILE"
      ACCEPT F1-NAME
      OPEN OLD F1
      ON EXCEPTION
      IF $$RES = "F"
           DISPLAY "FILE IN USE" SAMELINE
           GO TO AA100
      END
      DISPLAY "FILE NOT FOUND" SAMELINE
      GOTO AA100
      END
      .
      .
      .

 SECTION F1-ERR                 * INTERCEPT
      IF $$RES = "F" EXIT WITH 2    * ERROR F SPECIAL
      EXIT                      * ALL OTHERS NORMAL
 ENDPROG
```

Figure 1.6.4 - Example Program using an intercept routine

1.6.5 Exceptions Signalled by System Routines

A system routine always signals exception condition 1 if an I/O error is detected irrespective of whether the OPTION ERROR statement has

been coded in any FD supplied to the routine by the calling program. Different alphabetic settings of the result code, $$RES, distinguish the different types of I/O errors, as summarised in Table 1.6.1. The numeric settings of the code, shown in Table 1.6.1A, indicate the different circumstances in which the routines may signal exception condition 2. These circumstances are described in more detail, together with the $$RES values involved, in the documentation of each routine.

If the ON ERROR statement is coded in an FD supplied to a system routine, then the intercept routine involved will be called in the usual way if an I/O error is detected. If the intercept routine signals exception condition 2 to change an I/O error on an OPEN into a file condition, then the system routine will treat this as though file already exists (OPEN NEW) or file not found (OPEN OLD or SHARED) had been detected, although the alphabetic error code in $$RES will remain unchanged.

2. The Relative Sequential File Organisation

2.1 Relative Sequential Files

The records of an RS file can be output sequentially to the printer or accessed either at random or sequentially when the file is assigned to a direct access device. The records are fixed length and are numbered consecutively from 1 so that random access operations can retrieve data by record number.

2.1.1 Record Formats

Print files must be made up of print line records and, optionally, forms control records, as described in 1.5. The format of records appearing on any other type of RS file is totally under the control of the application program, except that the records of a file to be processed by Global AutoClerk should all begin with a two character type code. The first type character should not normally be an asterisk, since this is used to denote a logically deleted record:

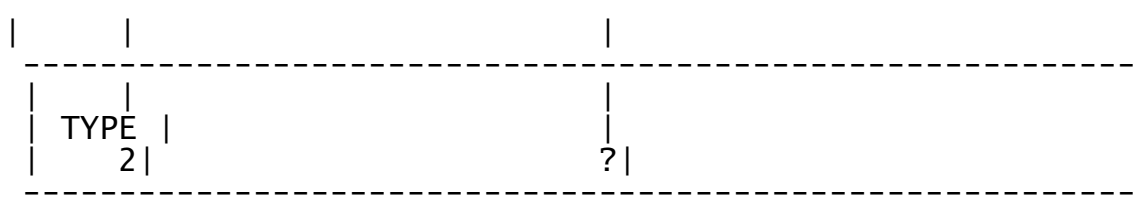


Figure 2.1.1 - AutoClerk Relative Sequential File Record Format

2.1.2 Specifying File Attributes

The attributes of an RS file, such as its unit-id, volume-id and file-id, are specified in its file definition (FD), coded in the data division. When creating a new file you supply the record length and the space to be allocated in the FD. You can also define a key area to contain the record number for use in random access READ and WRITE statements. Section 2.2 describes those parts of the file definition which are specific to the relative sequential file organisation, but the statements which are common to all organisations are defined in 1.2.

2.1.3 File Processing Statements

Nine procedure division statements are provided to enable an RS file to be processed. They are:

OPEN which must be executed prior to any other statement affecting the file;

WRITE NEXT to create the file initially or, in the case of a direct access file only, extend it by adding records at the end;

WRITE to update an existing record at random;

READ FIRST to read the very first record in the file;

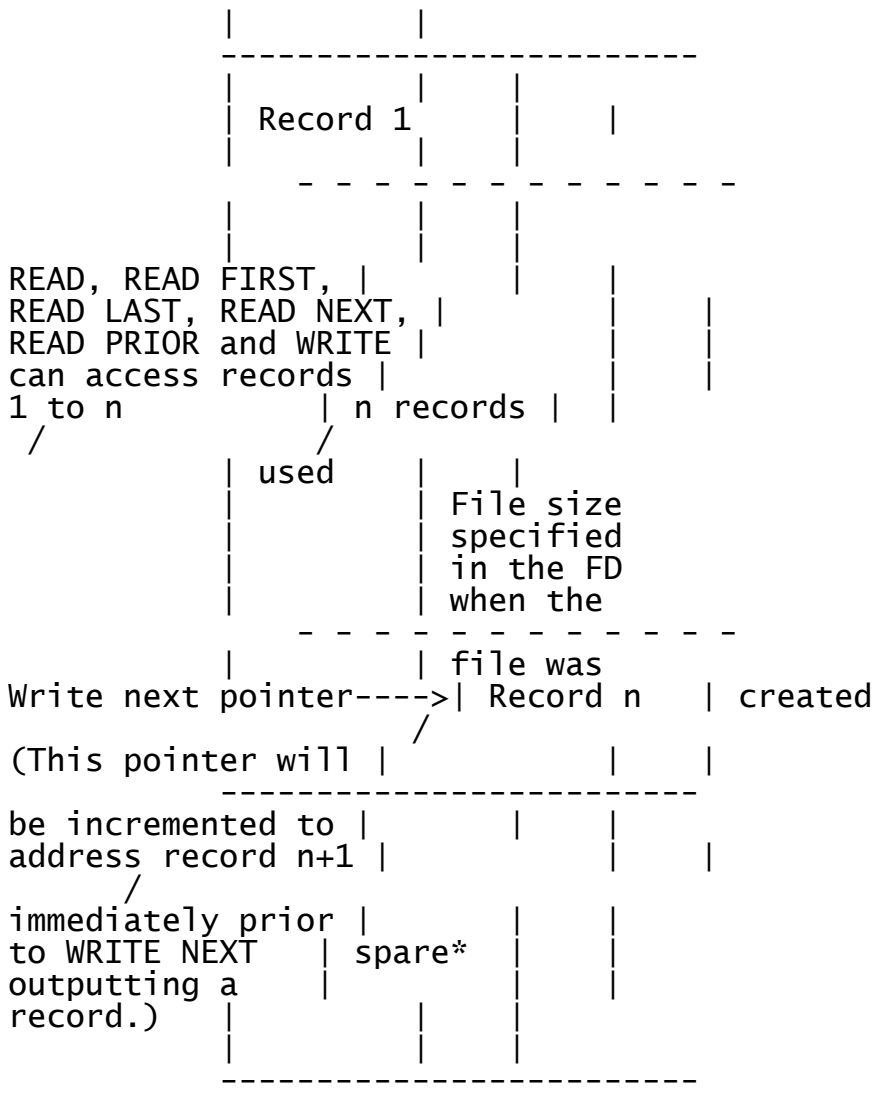
READ LAST to read the very last record in the file;

READ NEXT to read the next record sequentially;

READ PRIOR to read the previous record sequentially;

READ to read a record at random;

CLOSE to terminate processing of a file and, optionally, either delete it, or release any spare space it has for subsequent re-allocation.



* The spare space following the used part of the file can be returned to the volume and made available for re-allocation by CLOSE TRUNCATE.

Figure 2.1.4 - A Direct Access Relative Sequential File

Only OPEN, CLOSE and WRITE NEXT statements may be used on printer files.

When any of the statements is executed a file condition, signalled by exception condition 2, may arise as explained in 1.3.1. In addition, if OPTION ERROR is specified in the FD exception condition 1 will be generated should an irrecoverable I/O error occur. Therefore it is usual to follow each file processing statement with an ON EXCEPTION statement. If you do not, and an exception condition arises, your program will be terminated in error.

2.1.4 The Write Next Pointer

Throughout the life of a direct access relative sequential file the access method maintains a file pointer, the write next pointer, which determines the record affected by the WRITE NEXT operation and serves to delimit the used part of the file.

When an OPEN NEW statement completes satisfactorily the write next pointer is set to address an imaginary record immediately preceding the very first record of the file. Thereafter, whenever a WRITE NEXT operation takes place, the pointer is incremented to address the next record, and then that record is written. The write next pointer therefore addresses the last record of the file when there are any records at all. It is this pointer's value which is used in checking whether a READ or WRITE operation is within bounds.

The write next pointer is saved amongst other information retained on the volume whenever the file is closed so that it can be made available when the file is subsequently opened using OPEN OLD. Note that this means that if you fail to close a file, because of a programming error or machine failure, for example, then any new records created by WRITE NEXT since the file was opened will be lost, because the new value of the pointer will not have been saved.

Normally the write next pointer, incremented by WRITE NEXT, and indicating the used part of the file, is never reset. However, an option is provided so that you can re-use a work file by specifying that the write next pointer is to be reset to address the imaginary record preceding the very first record of the file when the file is opened by OPEN OLD. This allows work files to be allocated once by an initiation program and then used over and over again.

Figure 2.1.4 shows a relative sequential file after n records have been written using WRITE NEXT. The size specified in the file definition when the file was created allowed for more than n records so the used part of the file is followed by spare space. The spare space can be reserved for new records to be added to the end of the file by WRITE NEXT or it can be returned to the volume for re-allocation by a CLOSE statement using the TRUNCATE option.

READ, READ FIRST, READ LAST, READ NEXT, READ PRIOR and WRITE operations can access only the n records so far created.

2.2 The File Definition

The file definition for a relative sequential file is coded in either working storage or the linkage section as follows:

```
FD filename ORGANISATION RELATIVE-SEQUENTIAL
[ASSIGN TO UNIT unit-id FILE file-id [VOLUME volume-id]]
[KEY IS keyname]
[RECORD LENGTH IS length]
[SIZE IS size]
[OPTION ERROR [RESET]] | [OPTION RESET [ERROR]]
[ON ERROR intercept]
[BLOCK CONTAINS number CHARACTERS]
```

The FD establishes a special group data item, whose name is filename. The length of the item is 86 bytes, or 98 bytes if the BLOCK CONTAINS statement is coded. The quantities unit-id, file-id, volume-id,

keyname, length and size appear as subordinate items within this group and can, if need be, be referred to by the application program.

If it is possible to specify unit-id, file-id or volume-id before the program executes then the quantity should be coded as a character string in double quotes. If you can specify the size or length before the program executes, code the quantity as a numeric string. If any of these quantities is not known until run-time then a symbol must be coded for the quantity. This symbol will then label a level 02 item which the user program is responsible for initialising.

2.2.1 The Filename

The filename must be a symbol. It serves to label the file definition as explained in 1.2.1.

2.2.2 The ORGANISATION Clause

The ORGANISATION clause must be coded as shown, but you may use the abbreviations ORG and R-S instead of ORGANISATION and RELATIVE-SEQUENTIAL if you wish. It indicates that the file is relative sequential.

2.2.3 The ASSIGN Statement

Use of the ASSIGN statement, which is the same for any file organisation, is explained in section 1.2.

2.2.4 Optional Statement Placement

The KEY, RECORD LENGTH, SIZE, OPTION, ON ERROR and BLOCK statements are optional and may appear in any order following the ASSIGN statement.

2.2.5 The KEY Statement

The KEY statement is only required if records are to be accessed at random. Keyname is specified as a symbol and the statement:

```
    02 symbol PIC 9(9) COMP
```

is generated within the FD. The access method maintains the key field to contain the record number of the record last accessed: the field is set to zero when the file is opened.

The program must place the record number of the record it requires to access in the keyname field before executing a random READ or WRITE operation.

A successful READ NEXT operation increments the key field by one and then retrieves the record thus identified.

A successful READ PRIOR operation decrements the key field by one and then retrieves the record thus identified.

A successful READ FIRST or READ LAST operation returns the very first or very last record of the file respectively, and the key field is set appropriately.

A successful WRITE NEXT operation returns the record number of the record written in the key field.

2.2.6 The RECORD LENGTH Statement

The RECORD LENGTH statement is required only when creating a new file. The length is coded either as a positive integer between 1 and 32767 inclusive or as a symbol. When a symbol is used the statement:

```
02 symbol PIC 9(4) COMP
```

is generated within the FD. The program must place the record length in bytes in this field before executing an OPEN NEW statement.

A RECORD LENGTH statement supplying a symbol for the length may be used for an existing file. In this case when the OPEN statement completes satisfactorily the record length will be made available to the program in the generated field.

2.2.7 The SIZE Statement

The SIZE statement is required only when creating a new file. Its use, which is common to all file organisations, is explained in 1.2.8.

2.2.8 The OPTION and ON ERROR Statements

The OPTION statement allows you to specify one or more processing options according to the phrases you include following the OPTION verb. The phrases may be coded in any order and in any combination.

OPTION RESET is only used for an existing direct access file. It causes the open operation to reset the write next pointer to address an imaginary record immediately before the beginning of the file so that the file can be re-used. If RESET is not specified as an option OPEN OLD will restore the pointer from the value remembered when the file was last closed.

Note that you should not code OPTION RESET for a file which is opened to be shared by a number of co-operating multi-user jobs. If you issue an OPEN SHARED and there are already existing users of the file, the access method will not allow you to unilaterally reset it, and your program will be terminated in error.

OPTION ERROR should be coded only if you wish your program to regain control following an irrecoverable I/O error. ON ERROR processing should be coded if you wish to handle certain I/O errors specially. The processing of these statements is common to all file organisations, and is described in section 1.6 of this manual.

2.2.9 The BLOCK CONTAINS Statement

The BLOCK CONTAINS statement allows you to buffer sequential READ NEXT or WRITE NEXT operations on a direct access file, providing you have exclusive access to the file. This means you must have opened the file using OPEN NEW or OPEN OLD: buffering is inoperative on shared files although BLOCK CONTAINS still reserves buffer space following the file definitions. Note that WRITE NEXT should only be used with files opened for exclusive use, whereas READ NEXT can be used both for files opened exclusively and for those opened shared.

The block size you specify must be an integer between 1 and 32766 inclusive. It causes an uninitialised area of the size specified (rounded up, if necessary, to the nearest multiple of two bytes) to be reserved immediately following the file definition. The statement must therefore be used with care in environments where main storage is scarce. Buffering is not used when a file is written to a real printer, so the BLOCK CONTAINS statement should only be coded in the

FD for such a file if it is necessary to optimise performance in those cases where the file is assigned to a direct access device.

In the remainder of this discussion we assume that you are considering using the statement to improve performance. In this case the block size specified should always be at least twice the record size, as otherwise no blocking will be possible and it would be better to omit the statement altogether. Normally the number should be a multiple of the record length, but this is not essential and it may not be possible if the record length is not known until run-time.

Buffering is only performed for successive READ NEXT or successive WRITE NEXT operations. Other operations can be performed on a blocked file but these will not benefit from the blocking: indeed specifying blocking for a file which is accessed mainly at random might possibly degrade performance slightly. Hence you are recommended only to use the BLOCK CONTAINS statement on files which are mainly accessed sequentially.

Note that the block size is a property of the program accessing the file, and not of the file itself. The same file may be accessed with different block sizes in different programs, or accessed without blocking in others.

Note also that RSAM blocking is not available for the memory page version of RSAM.

2.3 The OPEN Statement

The OPEN statement is coded:

```
OPEN type filename
```

where type is the word NEW, OLD or SHARED and filename identifies the relative sequential file definition. An OPEN statement must be executed before any other operation affecting the file. If an OPEN is attempted but the FD is already open your program will be terminated in error.

OPEN NEW is used to create an RS file. OPEN OLD obtains exclusive access to an existing file and OPEN SHARED allows co-operating jobs running under multi-user System Manager to share a direct access RS file. (The features of the open operation which are common to all file organisations, such as volume-id checking, are described in detail in section 1.3.2.)

2.3.1 File Conditions

When OPEN NEW is used to access a direct access device, the System Manager checks to see whether a file with the same file-id as that specified in the FD is already present on the volume, and signals file already exists should this be the case.

When an OPEN OLD or OPEN SHARED operation accesses a direct access device, the System Manager checks to see whether a file with relative sequential organisation and the same file-id as that specified by the FD is present on the volume. If this is not the case file not found ($$RES = "3") or wrong type ($$RES = "1") is signalled.

A file condition cannot arise when opening a file assigned to a printer.

2.3.2 Successful Completion

Providing no file condition or irrecoverable I/O error occurs, an OPEN NEW results in the System Manager allocating the amount of space indicated by the FD's SIZE statement (or its absence). For a direct access file the write next pointer will be set to address an imaginary record immediately preceding the very first record of the file so that a subsequent WRITE NEXT operation will create the first record.

When an OPEN OLD or OPEN SHARED statement completes successfully the file size and record length are returned in the FD and can be accessed by the user program if SIZE and RECORD LENGTH statements with the symbol option were coded. For direct access files, the write next pointer is restored from the value saved in the file label, except if OPTION RESET is coded, in which case the pointer is set to address an imaginary record immediately preceding the first record of the file, so that the file can be re-used.

2.4 The WRITE NEXT Statement

WRITE NEXT is used to create a new RS file, or extend an existing direct access file by writing a new record at its end. It is coded:

```
WRITE NEXT filename FROM A
```

Here filename identifies the relative sequential file definition and A is a simple or indexed variable, or a literal. If a WRITE NEXT is attempted on an FD which is not already open the program will be terminated in error.

2.4.1 File Conditions

The file space exhausted condition will be signalled if there is insufficient space remaining in the file to write the new record. When outputting to a printer an alignment condition will occur if the record written was a forms control record asking the operator to confirm whether or not the paper was aligned satisfactorily, and he or she replied N to the resulting prompt, indicating that the alignment pattern is to be reprinted (see 1.5.5).

2.4.2 Successful Completion

If no file condition or irrecoverable I/O error occurs WRITE NEXT will increment the write next pointer and transfer bytes from A to the record so identified. The number of bytes transferred will not depend on the picture clause associated with A, but will be equal to the record length of the file as defined by the FD's RECORD LENGTH statement when the file was created.

The record number of the record written will be returned in the key field and can be accessed by the user program if the KEY statement was coded in the file definition.

2.5 The WRITE Statement

WRITE is used to replace a record initially created by WRITE NEXT. It is coded:

```
WRITE filename FROM A
```

Here filename identifies the relative sequential file definition and A is a simple or indexed variable, or a literal. If WRITE is attempted on an FD which is not already open the program will be terminated in error.

Do not use WRITE to send records to a printer.

2.5.1 Establishing the Key

If WRITE is to be used as a random access operation the FD must contain a KEY statement coded as:

```
KEY IS symbol
```

which causes:

```
02 symbol PIC 9(9) COMP
```

to be generated. The program must then move the record number of the record it is required to replace into this field before executing the WRITE statement.

If a KEY statement is not coded in the file definition the FD still contains an internal key field which is set to zero when the file is opened and incremented by READ NEXT. In this case the WRITE statement can be used to update the record previously retrieved by READ NEXT.

2.5.2 File Conditions

The file boundary violation condition will be signalled if the record number supplied in the key field lies outside the used part of the file delimited by the write next pointer.

2.5.3 Successful Completion

Providing no file condition or irrecoverable I/O error occurs, WRITE will transfer bytes from A to the specified record. The number of bytes transferred does not depend on the picture clause associated with A, but will be equal to the record length of the file as defined by the FD's RECORD LENGTH statement when the file was created.

2.6 The READ FIRST and READ LAST Statements

READ FIRST is used to retrieve the very first record in a file. It is coded:

```
READ FIRST filename INTO A
```

READ LAST is used to retrieve the very last record in a file. It is coded:

```
READ LAST filename INTO A
```

Here filename identifies the relative sequential file definition and A is a simple or indexed variable. If a READ FIRST or READ LAST is attempted on an FD which is not already open the program will be terminated in error.

2.6.1 File Conditions

The end of file condition will be signalled by READ FIRST if there are no records in the file, as delimited by the write next pointer. The

start of file condition will be signalled by READ LAST if there are no records in the file.

2.6.2 Successful Completion

Providing no file condition or irrecoverable I/O error occurs, the key field will be set to address the appropriate record (record 1 for READ FIRST, the last record number for READ LAST) and bytes will then be transferred from the record thus identified to A. The number of bytes transferred will not depend on the picture clause associated with A, but will be equal to the record length of the file, as defined by the FD's RECORD LENGTH statement when the file was created.

2.7 The READ NEXT and READ PRIOR Statements

READ NEXT is used to process part or all of a file sequentially in the order of the records. It is coded:

```
READ NEXT filename INTO A
```

READ PRIOR is used to process part or all of a file sequentially in reverse order of the records. It is coded:

```
READ PRIOR filename INTO A
```

Here filename identifies the relative sequential file definition and A is a simple or indexed variable. If a READ NEXT is attempted on an FD which is not already open the program will be terminated in error.

2.7.1 File Conditions

The end of file condition will be signalled on attempting to read past of the used part of the file using READ NEXT. For a direct access file this is delimited by the write next pointer. The start of file condition will be signalled on attempting to read the record before the first record on the file using READ PRIOR.

2.7.2 Successful Completion

Providing no file condition or irrecoverable I/O error occurs, READ NEXT will increment the record number in the key field by one, and READ PRIOR will decrement the record number in the key field by one. Bytes will then be transferred from the record thus identified to A. The number of bytes transferred will not depend on the picture clause associated with A, but will be equal to the record length of the file, as defined by the FD's RECORD LENGTH statement when the file was created.

2.7.3 Programming Note

If a KEY statement was not specified in the file definition the FD still contains an internal key field which is set to 0 when the file is opened. In this case READ NEXT can be used to read sequentially through the entire file; WRITE updates the last record thus retrieved; and READ rereads it. READ PRIOR used immediately after opening such a file would return the start of file condition, although it could be used after READ NEXT to retrieve a preceding record.

When a KEY statement is specified a READ or WRITE can be used to position the file at any point. READ NEXT will continue sequential processing from the record following the one retrieved by READ or output by WRITE, and READ PRIOR will continue sequential processing from the preceding record.

The WRITE NEXT statement updates the key field to address the last record of the file, so a READ NEXT following WRITE NEXT will be suppressed with a file operation exception indicating end of file.

2.8 The READ Statement

READ is used to retrieve a record at random or reread a record retrieved by READ NEXT. It is coded:

```
READ filename INTO A
```

Here filename identifies the relative sequential file definition and A is a simple or indexed variable. If READ is attempted on an FD which is not already open the program will be terminated in error.

2.8.1 Establishing the Key

If READ is to be used as a random access operation the FD must contain a KEY statement coded as:

```
KEY IS symbol
```

which causes:

```
02 symbol PIC 9(9) COMP
```

to be generated. The program must then move the record number of the record it is required to retrieve to this field before executing the READ statement.

If a KEY statement is not coded in the file definition the FD still contains an internal key field which is set to zero when the file is opened and incremented by READ NEXT or set by WRITE NEXT. In this case the only use of the READ statement is to reread the record previously retrieved by READ NEXT or written by WRITE NEXT.

2.8.2 File Conditions

The file boundary violation condition will be signalled if the record number supplied in the key field lies outside the used part of the file delimited by the write next pointer.

2.8.3 Successful Completion

Providing no file condition or irrecoverable I/O error occurs, READ will transfer bytes from the specified record to A. The number of bytes transferred does not depend on the picture clause associated with A, but will be equal to the record length of the file as defined by the FD's RECORD LENGTH statement when the file was created.

2.9 The CLOSE Statement

CLOSE must be used to terminate the processing of a file. It is coded:

```
CLOSE filename [TRUNCATE|DELETE]
```

where filename identifies the relative sequential file definition.

2.9.1 Standard Processing

CLOSE always completes any outstanding I/O operations on the file and returns the FD to the status it possessed prior to being opened.

Following CLOSE, the FD can be re-opened for the same, or a different, relative sequential file by a subsequent OPEN statement.

2.9.2 File Conditions

If a CLOSE is attempted on an FD which is not already open then the file not open condition will be signalled.

2.9.3 Truncation

The TRUNCATE phrase is ignored if the file is assigned to a printer. For direct access files it instructs the System Manager to return any spare space at the end of the extent to the volume so that it can be re-allocated.

2.9.4 Deletion

The DELETE phrase causes all the space occupied by a direct access file to be returned to the volume and its file-id to be erased from the directory. Thus following a CLOSE DELETE the affected direct access file no longer exists. However, the DELETE option has no effect on a printer.

2.9.5 Programming Notes

If you forget to close a print file in some circumstances the last few lines written by the program will not be output.

If you fail to close a direct access file any new records created by WRITE NEXT since the time it was opened may be lost. This is because the updated write next pointer is only preserved on the volume by the CLOSE or CLOSE TRUNCATE operations. If you are using WRITE NEXT to infrequently add records to the end of a relative sequential file during interactive working you should consider "checkpointing" the file by closing and re-opening it following each addition.

A WRITE operation, used to replace an existing record, is effected immediately so that a file which is only updated by WRITE's should normally remain consistent, even if it is not closed. This feature is intended to protect files which are updated interactively from damage in the event of machine failure: programs should always close the files they use.

2.10 Memory paged RSAM

This version of RSAM has most of the RSAM code within a System Manager memory page and therefore requires less space within a program. It cannot be used under pre-V6.2 Global System Manager.

Paged RSAM operates in exactly the same way as non-paged RSAM except that blocking is not allowed. No changes are required to any code using standard RSAM, all that is needed is that C.$PAGES is linked into your program before C.$APF and C.$MCOB ensuring that module AR$Z is linked in preference to module AR$B (see the section on memory paged subroutines in the System Subroutines manual).

2.11 The Relative Sequential Access Method for C-ISAM or Global files

This version of the Relative Sequential Access Method is only available under V8.1 or later System Manager and can be used to access both standard Global RSAM files and C-ISAM files. It is advised that you have a C-ISAM programmers manual available.

An RSAM file for C-ISAM consists of a Global schema file, containing various information about the C-ISAM file and created using RCBUILD and $SETIRU (see chapters 13 and 14), and a C-ISAM database.

2.11.1 Record Format

The record format expected is as described in section 2.1.1. The record format and fields must be in the Global. If a C-ISAM database is being accessed records will be converted to the C-ISAM format required, as specified within the schema file by the record conversion table, by the access method.

2.11.2 Specifying File Attributes

The attributes for the RS file, such as its unit-id, volume-id and file-id is specified in its file definition (FD), coded in the data division. These attributes must be either those of the Global relative-sequential file or for the C-ISAM schema file.

2.11.3 The File Definition

The file definition is as described in section 2.2 but the BLOCK contains statement is not allowed and buffer handling is not supported.

2.11.4 The File Processing Statements

The file processing statements are as described in section 2.1.3 and section 2.5 to 2.10 with the exception that OPEN NEW operations are not supported for C-ISAM files. If the access method receives a call for an OPEN NEW operation it will assume a System Manager relative sequential file is required and create a new Global RSAM file if possible.

If an I/O error occurs on the C-ISAM file the Unix error code will be returned in the system variable $$CRES, a PIC 9(9) COMP field. For information about the Unix error code see your C-ISAM manual from INFORMIX or Unix Manuals from your operating system supplier.

2.11.5 Programming Notes

The RSAM for Global or C-ISAM access method is a pageable routine (See System Subroutines manual for details) and its interface routine must be linked specially. You must include the following line in the $LINK dialogue before linking either C.$PAGES, C.$APF or C.$MCOB:

```
$44 LINK:C.$PAGES/AY$Z UNIT:$S
```

The routine will appear on the link map with program name AY$Z. Note that if you do not specifically link this routine but just link in C.$PAGES, then the pageable version of RSAM for System Manager files only, will be included (AR$Z) instead.

All locking of the relative sequential file must be done on the stub file within System Manager. This does mean that the C-ISAM database will not be locked from other Unix applications and therefore the C-ISAM file must not be accessed by other Unix application at the same time as it is being accessed by Global System Manager applications.

3. The Indexed Sequential File Organisation

3.1 Indexed Sequential Files

A Global IS file consists of fixed length data records and an index. Each data record contains a unique, fixed length, record key located in byte 5 onwards. The file processing statements OPEN, READ, READ NEXT, WRITE, REWRITE and CLOSE are used in conjunction with a file definition with ORGANISATION INDEXED-SEQUENTIAL to process an existing indexed sequential file:

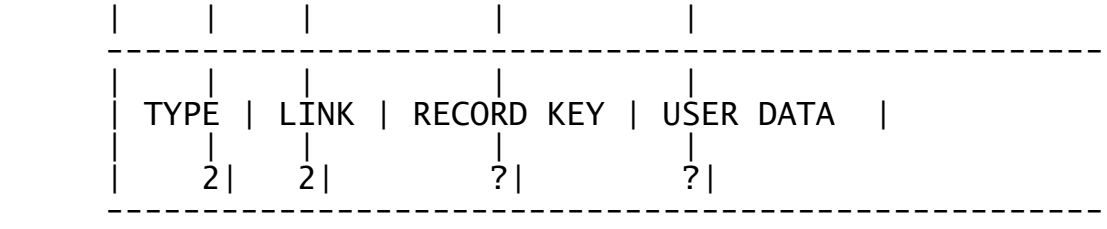
OPEN	must be executed prior to any other statement affecting the file;
READ	is used to retrieve a record at random, given its record key;
READ NEXT	retrieves the record whose key is next in collating sequence compared with the key last accessed. The operation can be used to process all or part of the file in record key sequence;
WRITE	is used to either update an existing record or insert a new record in sequence;
REWRITE	is used to update the record last accessed. It is more efficient than WRITE;
CLOSE	must be issued to terminate file processing.

IS files can only be created and maintained on direct access devices.

When any of the statements is executed a file condition, signalled by exception condition 2, may arise as explained in 1.3.1. In addition, if OPTION ERROR is specified in the FD, exception condition 1 will be generated should an irrecoverable I/O error occur. Therefore it is usual to follow each file processing statement with an ON EXCEPTION statement. If you do not, and an exception condition arises, your program will be terminated in error.

3.1.1 Data Record Format

The data records of an indexed sequential file consist of a two-character type code followed by a two-byte link field, the key and then any remaining user data:



The type may be used by AutoClerk, where it distinguishes the different sorts of records that make up the file. The first type character should not normally be an asterisk, since this used to denote a logically deleted record as explained in 3.1.4.

The link field is used for chaining new records which have been inserted into the correct sequence in the file. This field should not be used by the application programmer.

The record key is the value upon which the file is sequenced and indexed. For sequencing purposes it is treated by the access method as a character variable. The first byte of the key must not be high values. In practice record keys are normally defined to be character or positive computational items, or are contiguous groups of such items.

The record key, which must always begin in the fifth byte of each record, is followed by any amount of user data.

3.1.2 Creating a Global Indexed Sequential File

There are two ways of creating a Global indexed sequential file. Either you can start by building an empty file and then add to it, or you can convert a Global RS file to Global indexed sequential form. In both cases you create the IS file either by using the file conversion command ($CONV) interactively or by invoking the CONV$ conversion routine, described in section 9.1, under program control.

When an empty file is used it consists of a small prime data area and index (since the only data is one dummy record) and a much larger overflow area in which all the additions will be chained together. Unless you can arrange to add the records in descending key sequence, you will have to re-organise the file frequently, as explained in 3.1.5, to achieve acceptable performance. In any case you must re-organise before using random access operations on the file.

If you decide to create your IS file from an existing relative sequential file, you may avoid the need for early reorganisations. The RS file must contain records of the form described above. The contents of the link field are immaterial, but the records must be arranged in ascending collating sequence of record key, with no duplicates. When CONV$ or $CONV is used to produce an indexed sequential file from it the new file will consist of:

- A prime data area containing the original records;
- an index area, which immediately follows the prime data;
- an overflow area, following the index area and occupying the remainder of the space allocated to the file.

The index is read-only as far as the file operations are concerned. The overflow area is initially empty. A record is written to it whenever a WRITE for a record with a key not already present in the file takes place.

The space required for the index depends on the number of prime data records and the key length. The overflow area size depends on the maximum number of insertions expected before the file is next reorganised. The calculation command ($CALC) and system routine (CALC$) allow you to calculate the space requirements given these parameters.

The original relative sequential file is not modified in any way by the conversion process.

3.1.3 Specifying File Attributes

To process an existing indexed sequential file you specify attributes such as its unit-id, volume-id and file-id in a file definition (FD) coded in the data division. Section 3.2 below describes those parts of the FD which are specific to the indexed sequential file organisation, but the statements that are common to all organisations are defined in 5.2.

3.1.4 Logical Deletion of Records

No processing statements are provided to allow you to delete records from an indexed sequential file. Instead the convention is adopted that records whose type code begins with an asterisk, called logically deleted records, will be removed from the file when it is reorganised (with the exception of records with type code "*!", special deleted records used for pre-allocation, as explained in 3.1.9).

If you code the OPTION IGNORE statement in the file definition, then any deleted records will be ignored, as if they were not present. If you omit OPTION IGNORE then the deleted records are treated as normal records, and will be returned by READ and READ NEXT statements. This may be useful if you wish to reinstate an accidentally deleted record.

3.1.5 Reorganising a Global Indexed Sequential File

As more and more records are added to an indexed sequential file, overflow chains become longer, degrading performance, and the overflow area fills up. If logical deletions have taken place, unwanted records will still be physically occupying valuable file space. It may eventually become necessary to rearrange the file so that all records which have not been logically deleted occupy a new prime data area and the overflow area is empty. This process is termed reorganisation.

Reorganisation is accomplished either by using the file conversion command ($CONV) interactively, or by developing an application program to do the job, in which case it must use the CONV$ conversion routine as described in 9.1. In either case a new indexed sequential file is produced from the old one. The new file's prime data area contains all non-deleted records from the old file and its overflow area is empty.

The original indexed sequential file is not modified in any way by this conversion process.

Note that if you simply require to extend the overflow area and are not concerned with restructuring the prime data area or removing logically deleted records, this can be accomplished by simply copying the file to a larger area using either the file utility command program ($F) interactively, or by developing an application program using the COPY$ routine described in 9.2. A copy operation is much faster than a conversion since the file transfer proceeds using very large blocks rather than record by record.

3.1.6 Programming Notes

The indexed sequential access method described in this chapter is designed to be simple, so as to minimise the amount of main storage required by the access method itself, and to use compact files. The method (using Global files) is most suitable for files with a small percentage of evenly distributed insertions. This is because if A and B are two adjacent keys in the prime data area of the file then all

insertions with a key value between A and B are chained onto the B record. A dummy high key record is added to the end of the file to control the chain of records with keys greater than the original highest.

For high performance (for Global files) try not to use updating strategies involving large numbers of insertions at the same point within a file. In particular, avoid inserting multiple records at one point for a single transaction. As a rule of thumb try to limit the longest overflow chain to about 10 records before reorganising. If new additions tend to be allocated sequential keys you may need to pre-allocate records as described in 3.1.9 below.

The maximum number of overflow records in a Global ISAM file is 32,767 and the maximum size key is 99 bytes. Normally keys should be made as small as is practical since this reduces the size and number of levels of the index.

The Global ISAM file index contains a number of 256-byte blocks arranged in levels. Each block contains a 252-byte key table capable of holding a number of key values. Call this number, which depends of course on the key length, n. Then the first level index contains one index block for every n records in the prime data area. The second level index contains one block for every n blocks of the first level index. Index levels are constructed one after another until the highest level index, consisting of just a single index block, is developed.

The structure of a Global ISAM file is fully described in Appendix A.

3.1.7 Performance Guidelines for Global files

A READ requires to access one index block from each level and then at least one data record. More accesses will take place if the required record is on an overflow chain.

WRITE is even more expensive than READ because even if the data record already exists it has to be both read and written. If the record is new it has to be written to the end of the overflow area and the relevant overflow chain maintained, which requires an additional read and write. When adding a large number of new records consider inserting them in reverse key sequence in the interests of greater efficiency, since by doing so you reduce the amount of overlay chain scanning to a minimum. Always reorganise an IS file following a substantial number of insertions.

REWRITE writes the data record previously accessed. Since the address of this record is remembered there is no need for REWRITE to search the index to find the record.

READ NEXT retrieves a single data record using information saved in the file definition.

READ NEXT is the most efficient operation, then REWRITE, then READ and finally WRITE. READ and WRITE performance will be severely degraded if long overflow chains develop. The performance of READ and WRITE can be improved by making the highest level index block resident by supplying a 256-byte storage area for it in the OPEN OLD statement.

3.1.8 Bulk Additions

At times you will need to add large numbers of new records to an indexed sequential file. In particular, this will occur during initial data take-on. If you simply add these records directly to the file, performance will rapidly degrade and you will need to reorganise at frequent intervals.

It is better to write all the new records to a relative sequential file. When creating a new indexed file, simply sort the records into key sequence and convert the file to indexed sequential format. If the records are to be added to an existing file, then they should be sorted into reverse key order, and then inserted using WRITE statements. When they have all been added, the indexed file must be reorganised. (Inserting new records in reverse key order avoids long scans through overflow chains which degrade performance.)

Note that if the number of additions is manageably small it may be simpler to sort the records into reverse key order manually and then insert them directly into the file using your normal update procedure.

3.1.9 Pre-allocating Keys

When files are keyed on some form of reference code, such as a policy or account number, new records are normally allocated sequential codes. This causes them to be inserted at the same point in the file, and so if more than 20 or 30 additions are made between reorganisations, access to the new records will become very slow.

The problems caused by sequential additions can be overcome by adding special deleted records to the file immediately before it is reorganised. The keys used for the records should be in the range next to be allocated. The record type should be set to the two characters "*!". This special value causes the access method to consider the record to be logically deleted but instructs the reorganisation process to copy it to the output file rather than discard it. This means that following reorganisation the IS file will already contain the new keys expected in the next batch of additions. As a result these insertions become updates of the existing special deleted records and the performance overhead associated with long overflow chains is avoided.

In practice, it is not necessary to preallocate every key, and we would recommend you add a special deleted record only for every fifth or tenth key. o ensure the pre-allocation program itself runs as fast as possible insert the *! records themselves in reverse key order.

3.2 The File Definition

The file definition for an indexed sequential file is coded in either working storage or the linkage section as follows:

```
FD filename ORGANISATION INDEXED-SEQUENTIAL
[ASSIGN TO UNIT unit-id FILE file-id [VOLUME volume-id]]
[KEY LENGTH IS keylength]
[RECORD LENGTH IS length]
[SIZE IS size]
[OPTION ERROR [IGNORE] | OPTION IGNORE [ERROR]]
[ON ERROR intercept]
```

The FD establishes a special group data item, 94 bytes in length, whose name is filename. The quantities unit-id, file-id, volume-id,

keylength, length and size appear as subordinate items within this group and can, if need be, be referred to by the application program.

If it is possible to specify unit-id, file-id or volume-id before the program executes then the quantity should be coded as a character string in quotes. If you can specify keylength, size or length before the program executes, code the quantity as a numeric string. If any of these quantities is not known until run-time then a symbol must be coded for the quantity. This symbol will then label a level 02 item which the user program is responsible for initialising.

3.2.1 The Filename

The filename must be a symbol. It serves to label the file definition, as explained in 1.2.1.

3.2.2 The ORGANISATION Clause

The organisation clause must be coded as shown, except that you may use the abbreviations ORG and I-S instead of ORGANISATION and INDEXED-SEQUENTIAL. It indicates that the file is indexed sequential.

3.2.3 The ASSIGN Statement

Use of the ASSIGN statement, which is the same for any file organisation, is explained in section 1.2.

3.2.4 Optional Statement Placement

The KEY LENGTH, RECORD LENGTH, SIZE, ON ERROR and OPTION statements are optional and may appear in any order following the ASSIGN statement.

3.2.5 The KEY LENGTH Statement

The KEY LENGTH statement is required only when creating a new indexed sequential file using the conversion utility. The keylength is coded either as an integer between 1 and 99, or as a symbol. When a symbol is used the statement:

```
02 symbol PIC 9(2) COMP
```

is generated within the FD. The program must then place the key length to be used in the field before the conversion utility is invoked.

The KEY LENGTH statement with the symbol option can also be used for an existing file. In this case when the OPEN statement completes satisfactorily the actual length of the key will be returned in this field.

3.2.6 The RECORD LENGTH Statement

The RECORD LENGTH statement need only be supplied if you require to determine the file's record length at run-time. You code:

```
RECORD LENGTH IS symbol
```

causing the statement:

```
02 symbol PIC 9(4) COMP
```

to be generated within the file definition. When the OPEN operation terminates successfully the file's record length will be returned to you in the field named symbol.

3.2.7 The SIZE Statement

The SIZE statement is required only when creating a new indexed sequential file. The size is coded either as an integer between 0 and 999999999 inclusive, or as a symbol. When a symbol is used the statement:

```
02 symbol PIC 9(9) COMP
```

is generated within the FD. The program must then place the size to be used in this field before the conversion routine is invoked. If the size is specified as zero, or the SIZE statement is omitted, the routine will allocate the new indexed sequential file exactly as much space as that occupied by the originating relative sequential file extent.

The SIZE statement with the symbol option can also be used for an existing file. In this case when the OPEN OLD statement completes satisfactorily the actual number of bytes allocated to the file will be returned in the generated field.

3.2.8 The OPTION and ON ERROR Statements

OPTION ERROR should be coded only if you wish your program to regain control following an irrecoverable I/O error. ON ERROR should be coded if you wish to handle certain I/O errors specially. The processing of these statements is common to all file organisations, and is covered in section 1.6 of this manual.

OPTION IGNORE should be coded if you want deleted records to be ignored. If it is, deleted records will never be returned by a READ or READ NEXT statement, but will be treated as if they were not present on the file.

3.3 The OPEN Statement

The OPEN statement is coded:

```
OPEN type filename [index-area]
```

where type is the word OLD or SHARED and filename identifies the indexed sequential file definition. The index-area is optional. When specified it is the name of a 256-byte work area in which the highest level index will be held whilst the file is open. You supply this parameter if you wish to improve the performance of READ and WRITE operations by eliminating one index level access. The index area cannot be used for any other purpose whilst the file using it remains open, otherwise unpredictable errors will occur.

An OPEN statement must be executed before any other file processing statement. If an OPEN is attempted but the FD is already open your program will be terminated in error.

OPEN NEW is not supported, because an indexed sequential file is always created using CONV$ or $CONV, as explained in 3.1.2. OPEN OLD obtains exclusive access to the file whilst OPEN SHARED allows co-operating jobs running under multi-user System Manager to share an indexed sequential file. (The features of the open operation which are common to all file organisations, such as volume-id checking, are described in detail in section 1.3.2).

3.3.1 File Conditions
The file not found ($$RES = "3") or wrong type ($$RES = "1") condition is signalled if a file with indexed sequential organisation and the same file-id as that specified in the FD is not present on the direct access volume.

3.3.2 Successful Completion
Providing no file condition or irrecoverable I/O error occurs, the OPEN OLD operation completes successfully. The file size, record length and key length are returned in the FD and made available to the user program if SIZE, RECORD LENGTH and KEY LENGTH statements with the symbol option were coded.

3.4 The WRITE Statement
WRITE is used to update a record whose key is already present on the file or to add a record with a new key. It is coded:

```
WRITE filename FROM A
```

Here filename identifies the file definition and A is a simple or indexed variable, or a literal. If WRITE is attempted on an FD which is not already open the program will be terminated in error.

3.4.1 File Conditions
If a record with the specified key is not already present on the file, ISAM attempts to write a new record at the end of the overflow area, and then chains it into the existing file to maintain the key sequence. The file space exhausted condition will be signalled if there is insufficient space in the overflow area to contain the new record. This condition can never occur when the record key is already present on the file since in this case the WRITE operation simply causes an existing record to be updated.

3.4.2 Successful Completion
Providing that no file condition or irrecoverable I/O error occurs, WRITE will either cause an overflow record to be created or an existing record to be modified. The number of bytes transferred from the record at A to direct access storage will not depend on the picture clause of A but will be equal to the record length of the file, defined when it was created.

3.4.3 Programming Notes
When the record to be written is the one last accessed from the file, the REWRITE statement can, and normally should, be coded instead of WRITE since it is much more efficient.

3.5 The REWRITE Statement
REWRITE is used to update the record last accessed from a file. It is coded:

```
REWRITE filename FROM A
```

Here filename identifies the indexed sequential file definition and A is a simple or indexed variable or literal. If REWRITE is attempted on an FD which is not already open the program will be terminated in error.

3.5.1 Key Value Checking
ISAM checks that the key value supplied in A is the same as the key value of the record to be updated. If this is not the case the file operation will be suppressed and the program terminated in error. The key cannot be changed by a REWRITE operation: to modify the key a WRITE must be used.

3.5.2 Successful Completion
Providing the key has not been erroneously modified, and no irrecoverable I/O error occurs, REWRITE will update the record last accessed. The number of bytes transferred from the record at A to direct access storage will not depend on the picture clause of A but will be equal to the record length of the file, defined when it was created.

3.5.3 Programming Notes
Although a WRITE can be used to update an existing record, REWRITE should normally be employed when the record to be updated is the one last accessed, since it is more efficient. This is because ISAM remembers the address of the record last accessed and it is unnecessary for REWRITE to search the index to find the record.

3.6 The READ Statement
READ is used to retrieve a record with a given key from the file. If the key is not present the operation attempts to retrieve the record whose key is immediately higher than the one specified.

The READ statement is coded:

```
READ filename INTO A
```

Here filename identifies the indexed sequential file definition and A is a simple or indexed variable. If READ is attempted on an FD which is not already open the program will be terminated in error.

3.6.1 Establishing the Key
The key must be placed in the standard position of the record area (the fifth byte onwards) before the READ statement is executed. The key value will be replaced by a different value if the record not found file condition occurs.

3.6.2 File Conditions
The record not found file condition will be signalled if a record with the key you have specified is not present on the file. In this case ISAM will retrieve the record whose key is next higher in collating sequence than the one you specified, if such a record exists.

If the key you supplied was greater than any key currently stored, the file condition will again be signalled but the dummy high record will be returned to you. This dummy record contains a type code of ASCII blanks, a key of high values, and low-values for data. Note that if the key you supply is present on the file, the System Manager simply returns the appropriate record to you without signalling a file condition.

3.6.3 Successful Completion

Providing no permanent I/O error occurs and a key greater than or equal to the one you specified exists on the file, READ will transfer bytes from the record thus identified to A. The number of bytes transferred does not depend on the picture clause associated with A, but will be equal to the record length of the file defined when it was created.

3.7 The READ NEXT Statement

READ NEXT is used to process part or all of a file sequentially. It is coded:

```
READ NEXT filename INTO A
```

Here filename identifies the indexed sequential file definition and A is a simple or indexed variable. If a READ NEXT is attempted on an FD which is not already open the program will be terminated in error.

3.7.1 The Record Retrieved

The record retrieved by READ NEXT depends on the previous file operation:

- A READ NEXT following OPEN will retrieve the first record of the file.
- A READ NEXT following a READ, READ NEXT, WRITE or REWRITE will retrieve the record immediately higher in collating sequence than the one just read or written. If there is no such record end of file will be signalled.

3.7.2 File Conditions

The end of file condition will be signalled in response to READ NEXT if the record last accessed was the record with the highest key, or if the last file operation was a READ which was returned the dummy record because the key it required was greater than any held on the file. When this condition occurs the dummy high record is returned. This dummy record contains a type code of ASCII blanks, a key of high values, and low-values for data.

3.7.3 Successful Completion

Providing no file condition or irrecoverable I/O error occurs, bytes will be transferred from the file to A. The number of bytes transferred will not depend on the picture clause associated with A, but will be equal to the record length of the file, defined when it was created.

3.8 The CLOSE Statement

CLOSE must be used to terminate the processing of a file. It is coded:

```
CLOSE filename [TRUNCATE|DELETE]
```

where filename identifies the indexed sequential file definition.

3.8.1 Standard Processing

CLOSE always completes any outstanding I/O operations on the file and returns the FD to the status it possessed prior to being opened. Following CLOSE, the FD can be re-opened for the same, or a different, indexed sequential file by a subsequent OPEN statement.

3.8.2 File Conditions
If the CLOSE is attempted on an FD which is not already open the file not open condition will be signalled, and no processing will take place.

3.8.3 Truncation
If the TRUNCATE phrase is coded, ISAM will return the unused part of the overflow area, if any, to the volume so that it can be re-allocated.

3.8.4 Deletion
If the DELETE phrase is coded all the space the file occupies is returned to the volume and its file-id is erased from the directory. Following a CLOSE DELETE the file no longer exists.

3.8.5 Programming Note
WRITE and REWRITE operations on an indexed sequential file are effected immediately and at open time the access method is able to determine the true end of the overflow area. This means that an indexed sequential file should remain consistent, even if you fail to close it. This feature is intended to protect files which are updated interactively from damage in event of machine failure: programs should always close the files they use.

3.9 Memory Paged ISAM
This version of ISAM has most of the access method code within a System Manager memory page and therefore requires less spaces within a program. It cannot be used on pre-V6.2 Global System Manager.

Paged ISAM operates in exactly the same way as non-paged ISAM. No changes are required to any code using standard ISAM. All that is needed is that the compilation library C.$PAGES found on the SYSDEV unit is linked into your program before libraries C.$APF and C.$MCOB, ensuring that module AI$Z is linked in preference to module AI$A (see section on memory pages in the Systems Subroutines Manual).

3.10 The Indexed Sequential Access Method for Global or C-ISAM files
This version of ISAM can only be run on V8.1 or later of Global System Manager. It can be used to access data records in standard Global ISAM files or records whose data is held in a C-ISAM database. If using C-ISAM files you should refer to your C-ISAM programmer's manual.

An ISAM file for C-ISAM consists of a Global schema file, containing various information about the C-ISAM file and created using RCBUILD and $SETIRU (see Chapter 13 and 14), and a C-ISAM database containing at least the index by which the file is to be accessed using ISAM.

3.10.1 Record Format
The record format expected is as described for the indexed sequential file organisation in section 3.1.1. The record format and fields must be in the System Manager format. If a C-ISAM database is being accessed they will be converted by the access method to the C-ISAM format required as specified within the schema files by the record conversion table.

3.10.2 Specifying File Attributes
The attributes for the IS file, such as its unit-id, volume-id and file-id is specified in its file definition (FD), coded in the data division. These attributes must be those of the Global indexed sequential file or for the C-ISAM schema file.

3.10.3 The File Definition
The file definition is as described in section 3.2.

3.10.4 The File Processing Statements
The file processing statements are as described in section 3.3 to 3.8.

If an I/O error occurs on the C-ISAM file the Unix error code will be returned in the system variable $$CRES, a PIC 9(9) COMP field. For information about the Unix error code see your C-ISAM manual from INFORMIX or Unix Manuals from your operating system supplier.

3.10.5 Programming Notes
The ISAM for C-ISAM files is a pageable routine (see System Subroutines manual for details) and its interface routine must be linked specially. You must include the following line in the $LINK dialogue:

```
$44 LINK:C.$PAGES/AY$Z UNIT:$S
```

The routine will appear on the link map with program name AY$Z. Note that if you do not specifically link this routine but just link in C.$PAGES then the pageable version of RSAM for System Manager files will be included (AI$Z) in preference.

Unlike Global ISAM, C-ISAM will only support a single record key structure. Provision has therefore been made for a header and a trailer (first and last) record, which may be logically part of the indexed sequential file, to be held in a separate C-ISAM file. You can specify the separate header and trailer when creating the stub file using $SETIRU.

All locking of the indexed sequential file must be done on the stub file within System Manager. This does mean that the C-ISAM file will not be locked from other Unix applications and therefore the C-ISAM file must no be accessed by other Unix application at the same time as it is being accessed by Global applications.

4. The Variable Length Record File Organisation

4.1 Variable Length Record Files

The records of a VL file can be accessed either at random or sequentially when the file resides on direct access storage. The bytes of a VL file are numbered consecutively, starting at zero, and records can be accessed at random by supplying their starting byte number as a key.

4.1.1 Record Format

Each record of a VL file begins with a two-character type code and a two-byte PIC 9(4) COMP field containing the length of the total record (including the type field). User data, which can be of any format, follows the length field:

```
 |    |    |                    |
 ------------------------------------------------------
 |    |    |                    |
 | TYPE | LENGTH |   USER DATA          |
 |    2|   2|                   ?|
 ------------------------------------------------------
```

4.1.2 Specifying File Attributes

The attributes of a variable length record file, such as its unit-id, volume-id and file-id, are specified in its file definition (FD), coded in the data division. When creating a new file you supply the space to be allocated in the FD. You can also define a key area to contain the starting byte number for use in random access READ and WRITE statements. Section 4.2 below describes those parts of the file definition which are specific to the variable length record file organisation, but the statements which are common to all organisations are defined in section 1.2.

4.1.3 File Processing Statements

Seven procedure division statements are provided to enable a VL file to be processed. They are:

OPEN which must be executed prior to any other statement affecting the file;

READ to read a record at random (or READ PHYSICAL);

READ FIRST to read the very first record in the file;

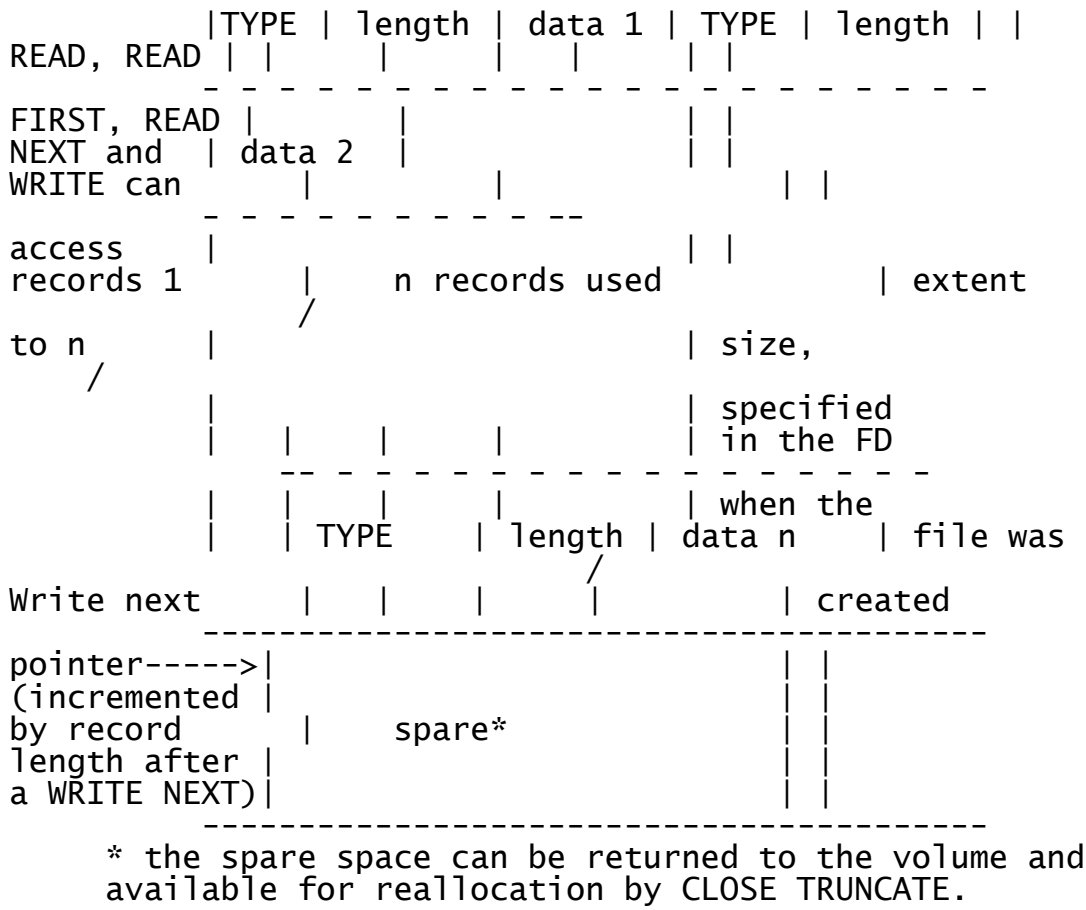
READ NEXT to read a record during sequential processing;

WRITE to update an existing record at random;

WRITE NEXT to create the file or extend it by writing a new record at its end;

CLOSE to terminate processing of a file.

```
| |  |     |   |     |
           ---------------------------------------
           | | |     |   |     | |
```



* the spare space can be returned to the volume and made available for reallocation by CLOSE TRUNCATE.

Figure 4.1.4 - A Direct Access Variable Length Record File

When any of the statements is executed a file condition, signalled by exception condition 2, may arise as explained in section 1.3.1. In addition if OPTION ERROR or ON ERROR is specified in the FD, an exception condition will be generated should an irrecoverable I/O error occur. Therefore it is usual to follow each file processing statement with an ON EXCEPTION statement. If you do not, and an exception condition arises, your program will be terminated in error.

4.1.4 The Write Next Pointer

Throughout the life of a direct access variable length record file VLAM maintains a file pointer, the write next pointer, which determines the record affected by the WRITE NEXT operation and serves to delimit the used part of the file.

When an OPEN NEW statement completes satisfactorily the write next pointer is set to zero. Thereafter, whenever a WRITE NEXT operation is successfully completed the write next pointer is incremented by the length of the record written. The write next pointer therefore always addresses the first unused byte on the file. It is this pointer's value which is used in checking whether a READ, READ FIRST, READ NEXT or WRITE operation is within bounds.

The write next pointer is saved amongst other information retained on the volume whenever the file is closed, so that it can be made available when the file is subsequently opened using OPEN OLD. Note that this means that if you fail to close a file, because of a programming error or machine failure, for example, then any new

records created by WRITE NEXT since the file was opened will be lost, because the new value of the pointer will not have been saved.

Figure 4.1.4 shows a variable length record file after n records have been written using WRITE NEXT. The size specified in the file definition when the file was created allowed for more records so the used part of the file is followed by spare space. The spare space can be reserved for new records to be added to the end of the file by WRITE NEXT or it can be returned to the volume for re-allocation by a CLOSE statement using the TRUNCATE option.

READ, READ FIRST, READ NEXT and WRITE operations can access only the n records so far created.

4.1.5 The ORGANISATION Statement

If a program uses a variable length record file, then the statement:

```
ORGANISATION OR$84 TYPE 4 EXTENSION 8
```

must be coded in the data division before the first FD or data declaration.

4.2 The File Definition

The file definition for a variable length record file is coded in either working storage or the linkage section as follows:

```
FD filename ORGANISATION OR$84
[ASSIGN TO UNIT unit-id FILE file-id [VOLUME volume-id]]
[KEY IS keyname]
[RECORD LENGTH IS length]
[SIZE IS size]
[OPTION ERROR]
[ON ERROR intercept]
```

The FD establishes a special group data item, 88 bytes in length, whose name is filename. The quantities unit-id, file-id, volume-id, keyname, length and size appear as subordinate items within this group and can, if need be, be referred to by the application program.

If it is possible to specify unit-id, file-id or volume-id before the program executes then the quantity should be coded as a character string in quotes. If you can specify the size or length before the program executes, code the quantity as a numeric string. If any of these quantities is not known until run-time then a symbol must be coded for the quantity. This symbol will then label a level 02 item which the user program is responsible for initialising.

4.2.1 The Filename

The filename must be a symbol. It serves to label the file definition, as explained in section 1.2.1.

4.2.2 The ORGANISATION Clause

The ORGANISATION clause must be coded as shown. It indicates that the file contains variable length records.

4.2.3 The ASSIGN Statement

Use of the ASSIGN statement, which is the same for any file organisation, is explained in section 1.2.

4.2.4 Optional Statement Placement
The KEY, RECORD LENGTH, SIZE, OPTION and ON ERROR statements are optional and may appear in any order following the ASSIGN statement.

4.2.5 The KEY Statement
The KEY statement is only required if records are to be accessed at random. Keyname is specified as a symbol and the statement:

```
02 symbol PIC 9(9) COMP
```

is generated within the FD.

VLAM maintains in the keyname field the byte number of the start of the record last accessed.

The program must place the byte number of the start of the record it requires to access in the keyname field before executing a random READ or WRITE operation.

A successful READ NEXT operation increments the keyname field by the length of the previous record accessed and then retrieves the record thus identified: if the previous operation on the file was an OPEN then the first record is read.

A successful WRITE NEXT operation returns the byte number of the start of the record written in the keyname field.

4.2.6 The RECORD LENGTH Statement
The RECORD LENGTH statement is only required when reading records from a file. It specifies the length of the area into which the record is being read, ie the maximum length record which can be read. If this length is fixed then it can be specified as an integer between 4 and 32767 inclusive, otherwise the RECORD LENGTH statement should specify a symbol and the length must be set up before a READ or READ NEXT statement is executed.

4.2.7 The SIZE Statement
The SIZE statement is required only when creating a new file. Its use, which is common to all file organisations, is explained in Section 1.2.8.

4.2.8 The OPTION and ON ERROR Statements
OPTION ERROR should be coded only if you wish your program to regain control following an irrecoverable I/O error. ON ERROR should be coded if you wish to handle certain I/O errors specially. The processing of these statements is common to all file organisations, and is covered in section 1.6 of this manual.

4.3 The OPEN Statement
The OPEN statement is coded:

```
OPEN type filename
```

where type is the word NEW, OLD or SHARED and filename identifies the variable length record file definition. An OPEN statement must be executed before any other operation affecting the file. If an OPEN is

attempted but the FD is already open your program will be terminated in error.

OPEN NEW is used to create a VL file. OPEN OLD obtains exclusive access to an existing file and OPEN SHARED allows co-operating jobs running under multi-user System Manager to share a direct access VL file. (The features of the open operation which are common to all file organisations, such as volume-id checking, are described in detail in Section 1.3.2.)

4.3.1 File Conditions

When an OPEN NEW is used to access a direct access device the System Manager will signal file already exists if a file with the same file-id as that specified in the FD is present.

When an OPEN OLD or OPEN SHARED statement is executed the System Manager checks to see whether a file with variable length record organisation and the same file-id as that specified by the FD is present on the volume. If this is not the case wrong type ($$RES = "1") or file not found ($$RES = "3") is signalled.

4.3.2 Successful Completion

Providing no file condition or irrecoverable I/O error occurs an OPEN NEW results in the System Manager allocating the amount of space indicated by the FD's SIZE statement (or its absence). For a direct access file the write next pointer will be set to address the first byte so that a subsequent WRITE NEXT operation will create the first record.

When an OPEN OLD or OPEN SHARED statement completes successfully the file size is returned in the FD and can be accessed by the user program if a SIZE statement with the symbol option was coded.

4.4 The WRITE NEXT Statement

WRITE NEXT is used to create the records of a new VL file, or extend an existing direct access VL file by writing a new record at its end. It is coded:

```
WRITE NEXT filename FROM A
```

Here filename identifies the variable length record file definition and A is a simple or indexed variable. If a WRITE NEXT is attempted on an FD which is not already open, the program will be terminated in error.

4.4.1 File Conditions

The file space exhausted condition will be signalled if there is insufficient space remaining in the extent to write the new record.

4.4.2 Record Length Checking

The length of the record to be written, given by bytes 3 and 4 of the record, must be in the range 4 to 32767 otherwise the program will be terminated in error.

4.4.3 Successful Completion

Providing a valid record length is supplied and no file condition or irrecoverable I/O error occurs, WRITE NEXT will transfer the number of

bytes given by the length field in A from the record area A to the file address given by the write next pointer. The write next pointer will then be incremented by the length of the record written.

The starting byte number of the record written will be returned in the keyname field and can be accessed by the user program if the KEY statement was coded in the file definition.

4.5 The WRITE Statement

WRITE is used to replace a record initially created by WRITE NEXT. It is coded:

```
WRITE filename FROM A
```

Here filename identifies the variable length record file definition and A is a simple or indexed variable. If WRITE is attempted on an FD which is not already open the program will be terminated in error.

4.5.1 Establishing the Key

If WRITE is to be used as a random access operation the FD must contain a KEY statement coded as:

```
KEY IS symbol
```

which causes:

```
02 symbol PIC 9(9) COMP
```

to be generated. The program must then move the byte number of the start of the record it is required to replace into this field before executing the WRITE statement.

If a KEY statement is not coded in the file definition the FD still contains an internal key field which is set to zero when the file is opened and incremented by READ NEXT. In this case the WRITE statement can be used to update the record previously retrieved by READ NEXT.

4.5.2 File Conditions

The file boundary violation condition will be signalled if the byte number supplied in the keyname field lies outside the used part of the file delimited by the write next pointer.

4.5.3 Record Length Checking

If the length of the record to be written is not the same as the length of the record it is to replace then the program will be terminated in error.

4.5.4 Successful Completion

Providing no file condition or irrecoverable I/O error occurs and the record lengths are the same then the existing record will be replaced in its entirety by the new record. The length of the record does not depend on the picture clause of A, only on the length in bytes 3 and 4 of A.

4.5.5 Programming Note

A WRITE operation will always complete the updating of the file before returning control, so that even if the program subsequently fails the file will have been updated.

4.6 The READ FIRST Statement

READ FIRST is used to read the very first record in the file. It is coded:

```
READ FIRST filename INTO A
```

Here filename identifies the variable length record file definition, and A is a simple or indexed variable. If READ FIRST is attempted on an FD which is not already open the program will be terminated in error.

4.6.1 File Conditions

The end of file condition will be signalled if the file is empty (contains no records).

4.6.2 Record Length Checking

If the length of the record to be read, given by bytes 3 and 4 of the record, is greater than the maximum length of A, given by the record length specified in the FD, then the program will be terminated in error.

4.6.3 Successful Completion

Provided no file condition or irrecoverable I/O error occurs, and the record is not longer than the record length in the FD, then READ FIRST will set the key to address the very first record in the file (a value of zero) and transfer this record to A. The number of bytes transferred is given by the length field in bytes 3 and 4 of the record area.

4.7 The READ NEXT Statement

READ NEXT is used to process part or all of a file sequentially. It is coded:

```
READ NEXT filename INTO A
```

Here filename identifies the variable length record file definition and A is a simple or indexed variable. If a READ NEXT is attempted on an FD which is not already open the program will be terminated in error.

4.7.1 File Conditions

The end of file condition will be signalled on attempting to read past the used part of the file. For a direct access file this is delimited by the write next pointer.

4.7.2 Record Length Checking

If the length of the record to be read, given by bytes 3 and 4 of the record, is greater than the maximum length of A, given by the record length specified in the FD, then the program will be terminated in error.

4.7.3 Successful Completion

Provided no file condition or irrecoverable I/O error occurs, and the record is not longer than the record length in the FD, then READ NEXT will increment the key by the length of the previous record accessed (if any) and transfer the record thus identified to A. The number of bytes transferred is given by the length field in bytes 3 and 4 of the record area.

4.7.4 Programming Note

If a KEY statement was not specified in the file definition the FD still contains an internal key field which is set to 0 when the file is opened. In this case READ NEXT can be used to read sequentially through the entire file; WRITE updates the last record thus retrieved; and READ rereads it.

When a KEY statement is specified a READ or WRITE can be used to position the file at any point. READ NEXT will continue sequential processing from the record following the one retrieved by READ or output by WRITE.

The WRITE NEXT statement updates the key field to address the last record of the file, so a READ NEXT following WRITE NEXT will be suppressed with a file operation exception indicating end of file.

4.8 The READ Statement

READ is used to retrieve a record at random or reread a record retrieved by READ NEXT. It is coded:

```
READ filename INTO A
```

Here filename identifies the variable length record file definition and A is a simple or indexed variable. If READ is attempted on an FD which is not already open the program will be terminated in error.

4.8.1 Establishing the Key

If READ is to be used as a random access operation the FD must contain a KEY statement coded as:

```
KEY IS symbol
```

which causes:

```
02 symbol PIC 9(9) COMP
```

to be generated. The program must then move the byte number of the start of the record it is required to retrieve to this field before executing the READ statement.

If a KEY statement is not coded in the file definition the FD still contains an internal key field which is set to zero when the file is opened and incremented by READ NEXT or set by WRITE NEXT. In this case the only use of the READ statement is to reread the record previously retrieved by READ NEXT or written by WRITE NEXT.

4.8.2 File Conditions

The file boundary violation condition will be signalled if the byte number supplied in the keyname field lies outside the used part of the file delimited by the write next pointer.

4.8.3 Record Length Checking
If the length of the record to be read, given by bytes 3 and 4 of the record, is greater than the length of A, given by the record length specified in the FD, then the program will be terminated in error.

4.8.4 Successful Completion
Providing no file condition or irrecoverable I/O error occurs and the record is not longer than the record length in the FD, then READ will transfer to A the number of bytes given by bytes 3 and 4 of the record identified by the key.

4.8.5 The READ PHYSICAL Statement
As the key of a variable length record file is the byte offset of the record, the READ PHYSICAL statement is identical in function to the READ statement.

4.9 The CLOSE Statement
CLOSE must be used to terminate the processing of a file. It is coded:

```
CLOSE filename [TRUNCATE|DELETE]
```

where filename identifies the variable length record file definition.

4.9.1 Standard Processing
CLOSE always completes any outstanding I/O operations on the file and returns the FD to the status it possessed prior to being opened. Following CLOSE, the FD can be re-opened for the same, or a different, variable length record file by a subsequent OPEN NEW or OPEN OLD statement.

4.9.2 File Conditions
If the FD is not open then CLOSE will signal the file not open condition, and the FD will not be processed.

4.9.3 Truncation
If the file is assigned to a direct access device the TRUNCATE phrase instructs the System Manager to return any spare space at the end of the extent to the volume so that it can be re-allocated.

4.9.4 Deletion
The DELETE phrase causes all the space occupied by a direct access file to be returned to the volume and its file-id to be erased from the directory. Thus following a CLOSE DELETE the affected file no longer exists.

4.9.5 Programming Notes
If you fail to close a direct access file any new records created by WRITE NEXT since the time it was opened will be lost. This is because the updated write next pointer is only preserved on the volume by the CLOSE or CLOSE TRUNCATE operations. If you are using WRITE NEXT to infrequently add records to the end of a variable length record file during interactive working you should consider "checkpointing" the file by closing and re-opening it following each addition.

A WRITE operation, used to replace an existing record, is effected immediately so that a file which is only updated by WRITE operations should normally remain consistent, even if it is not closed. This

feature is intended to protect files which are updated interactively from damage in the event of machine failure: programs should always close the files they use.

5. The Text File Organisation

5.1 Text Files

The records of a TF file can be written or read sequentially when the file is assigned to a direct access device. It is also possible to read a record at random by supplying its starting character number. The file is treated as a contiguous string of ASCII characters numbered from zero.

5.1.1 Notation

In describing the non-graphic ASCII characters such as form-feed, vertical-tab and so on, in the examples within this chapter we use the short codes normally associated with those characters, as listed in Appendix A of the Global Cobol Language Manual. In particular:

NUL null, the character with a byte value of binary zero;

HT horizontal tab (the character normally referred to simply as tab, when the context distinguishes it from vertical tab);

VT vertical tab;

FF form feed;

CR carriage return;

LF line feed;

SP space.

5.1.2 File Format

A text file consists of a string of ASCII characters not including NUL. Each character occupies a single byte in which the senior, parity, bit is set to zero. End of file is indicated when a terminating NUL character is detected, or in the case of direct access text files only, at the end of the extent occupied by the file if no preceding NUL was found.

5.1.3 Line Format

A text file is considered to be made up of a number of contiguous lines of information. Each line except the first is separated from its predecessor by the newline sequence with which it begins. The newline sequence at the start of the very first line of the file may be omitted in some cases, but in every other case it must assume one of the following formats:

- a single VT character;
- a single FF character;
- CR or LF, or any combination of these two characters.

The line body consists of those characters following the newline sequence (if present). The body is delimited by the first character of the next line, or by end of file in the case of the last line. The body may be empty, in which case the line consists merely of a newline sequence.

The first significant character within a line is the first one which is neither VT, FF, CR, LF, HT or SP. It must obviously be within the line body, and it is possible that a line possesses no first significant character.

5.1.4 Record Format

When a program performs a normal text file read operation the information returned by the access method in its record area consists of a line immediately followed by a NUL character, which serves as a delimiter. When the program wishes to write a line it must set up the record area in the same way, providing the line information first, and then the NUL terminator.

Note that the PRIN$ system routine, described in section 15.1, can be used to convert a record read from a text file to a format which is suitable for printing or displaying on the screen. The routine converts the newline sequence into the appropriate print control byte (or eliminates it in the case of a display) and expands any horizontal tabs appearing within the line body.

5.1.5 Example

The following ASCII character string forms a text file, which is terminated by the NUL character:

```
     A B FF LF SP SP X Y CR LF LF R S T NUL
```

Table 5.1.5 below shows the four records which would be required to create this file. Identical records would be returned, in the order listed, were the file subsequently to be read sequentially.

		RECORD AREA CONTENTS		
		FIRST		
RECORD NUMBER	NEWLINE SEQUENCE	LINE BODY	TERMINATOR	SIGNIFICANT CHARACTER
1	none	A B	NUL	A
2	FF	none	NUL	none
3	LF	SP SP X Y	NUL	X
4	CR LF LF	R S T	NUL	R

Table 5.1.5 - Sample Text File Records

5.1.6 Specifying File Attributes

The attributes of a text file, such as its unit-id, volume-id and file-id, are specified in its file definition (FD), coded in the data division. When creating a new file you supply the space to be allocated in the FD. You can also define a key area to contain the character number for use by the random access READ operations. Section 5.2 below describes those parts of the file definition which are specific to the text file organisation, but the statements which are common to all organisations are described in section 1.2.

5.1.7 File Processing Statements

Six procedure division statements are provided to enable a TF file to be processed. They are:

OPEN which must be executed prior to any other statement affecting the file;

READ to read a record at random;

READ FIRST to read the very first record of the file;

READ NEXT to read a record during sequential processing;

WRITE NEXT to create the file or extend it by writing a new record at its end;

CLOSE to terminate processing of a file.

When any of the statements is executed a file condition, signalled by exception condition 2, may arise as explained in section 1.3.1. In addition a READ, READ FIRST or READ NEXT can suffer a record length exception, peculiar to text file handling. This exception, which sets $$COND to 3, indicates that the record the operation attempted to retrieve was longer than the record area reserved for it. Finally, of course, if OPTION ERROR or ON ERROR is specified in the FD, an exception condition will be generated should an irrecoverable I/O error occur, as explained in section 8.2. Therefore it is usual to follow each file processing statement with an ON EXCEPTION statement. If you do not, and an exception condition arises, your program will be terminated in error.

5.1.8 The Write Next Pointer

Throughout the life of a direct access text file TFAM maintains a file pointer, the write next pointer, which serves to delimit the used part of the file.

The write next pointer contains the character number of the null character terminating the file, unless the file completely fills the extent, in which case it addresses an imaginary byte following the file extent. When an OPEN NEW statement completes satisfactorily the write next pointer is set to zero. It is incremented by the length of the record written whenever a WRITE NEXT takes place.

The write next pointer is saved amongst other information retained on the volume whenever the file is closed so that it can be made available when the file is subsequently opened.

Note that this means that if you fail to close a file, because of a programming error or machine failure for example, then any new records

created by WRITE NEXT since the file was opened will be lost, because the new value of the pointer will not have been saved.

5.1.9 The ORGANISATION Statement

If a program uses text files which are write-only, then the following statement must be coded in the data division before the first FD or data declaration:

```
ORGANISATION OR$83 TYPE 3 EXTENSION 16
```

However, if READ, READ FIRST or READ NEXT statements are used on a text file, then the access method requires the standard 16-byte extension to be followed by an additional 512-byte buffer. If every text file FD used by the program is subject to READ, READ FIRST or READ NEXT statements then the simplest way of generating the extra buffer is to request a larger extension in the ORGANISATION statement by coding:

```
ORGANISATION OR$83 TYPE 3 EXTENSION 528
```

However, this method is wasteful on storage when some text file FDs are write-only, since these FDs are expanded with a 512-byte buffer which they do not require. For this reason, when some, but not all, text file FDs are write-only we recommend that you code an ORGANISATION statement requesting a 16-byte extension, and then explicitly declare a 512-byte buffer immediately following each text file FD supporting read operations. For example:

```
PROGRAM TEXTP
DATA DIVISION
ORGANISATION OR$83 TYPE 3 EXTENSION 16
.
.
FD INFIL ORGANISATION OR$83
ASSIGN TO UNIT "DSK" FILE "TXINFIL"
01 FILLER
02 FILLER PIC X(512) ** BUFFER
.
.
FD OUTFIL ORGANISATION OR$83
ASSIGN TO UNIT "DSK" FILE "TXOUTFIL"
RECORD LENGTH IS 72
** NO BUFFER
```

Here the INFIL FD, together with the buffer which immediately follows it, occupies 608 bytes whereas the OUTFIL FD only requires 96 bytes.

Note that if the CATA$ system routine is used to determine the volume-id and unit-id of a text file for which you have declared an explicit buffer, then the FD involved should either be the only file definition in the list passed to CATA$, or the last FD in the list.

This is because CATA$ uses the EXTENSION information supplied by the ORGANISATION clause to deduce the length of each file definition when scanning its list. If you have declared an explicit buffer the length used by CATA$ will be inaccurate, and the routine will be unable to find the next FD. This problem will not, of course, arise if the file definition in question is the last, or only, member of its list.

5.2 The File Definition

The file definition for a text file is coded in either working storage or the linkage section as follows:

```
FD filename ORGANISATION OR$83
[ASSIGN TO UNIT unit-id FILE file-id [VOLUME volume-id]]
[KEY IS keyname]
[RECORD LENGTH IS length]
[SIZE IS size]
[OPTION ERROR]
[ON ERROR intercept]
```

The FD establishes a special group data item whose name is filename. The quantities unit-id, file-id, volume-id, keyname, length and size appear as subordinate items within this group and can, if need be, be referred to by the application program. The size of the FD is either 96 or 608 bytes, depending on whether it is used for output only, and how the ORGANISATION statement has been coded. (See 1.1.9.)

If it is possible to specify unit-id, file-id or volume-id before the program executes then the quantity should be coded as a character string in quotes. If you can specify the size or length before the program executes, code the quantity as a numeric string. If any of these quantities is not known until run-time then a symbol must be coded for the quantity. This symbol will then label a level 02 item which the user program is responsible for initialising.

5.2.1 The Filename

The filename must be a symbol. It serves to label the file definition, as explained in section 1.2.1.

5.2.2 The ORGANISATION Clause

The ORGANISATION clause must be coded as shown. It indicates that the file is a text file.

5.2.3 The ASSIGN Statement

Use of the ASSIGN statement, which is the same for any file organisation, is explained in section 1.2.

5.2.4 Optional Statement Placement

The KEY, RECORD LENGTH, SIZE, OPTION and ON ERROR statements are optional and may appear in any order following the ASSIGN statement.

5.2.5 The KEY Statement

The KEY statement is only required if records are to be accessed at random. Keyname is specified as a symbol and the statement:

```
02 symbol PIC 9(9) COMP
```

is generated within the FD.

TFAM maintains the keyname field to contain the character number of the start of the line last accessed: the field is set to zero when the file is opened.

The program must place the character number of the start of the line it requires to access in the keyname field before executing a random READ operation.

A successful READ NEXT operation increments the keyname field by the length of the previous line read and then retrieves the record thus identified.

A successful WRITE NEXT operation returns the starting character number of the line written in the keyname field.

5.2.6 The RECORD LENGTH Statement

The RECORD LENGTH statement is only required when reading records from a text file. It specifies the length of the area into which the line is being read, ie the maximum length line, including the terminating NUL, which can be read. If this length is fixed then it can be specified as an integer between 2 and 257 inclusive, otherwise the RECORD LENGTH statement should specify a symbol and the length must be set up before a read statement is executed.

5.2.7 The SIZE Statement

The SIZE statement is required only when creating a new file. Its use, which is common to all file organisations, is explained in section 1.2.8.

5.2.8 The OPTION and ON ERROR Statements

OPTION ERROR should be coded only if you wish your program to regain control following an irrecoverable I/O error. ON ERROR should be coded if you wish to handle certain I/O errors specially. The processing of these statements is common to all file organisations, and is described in section 1.6 of this manual.

5.2.9 Additional Fields in the FD

A text file FD contains three additional fields which your program can access by including a redefinition of the FD of the following form:

```
01 TX REDEFINES filename
03 FILLER  PIC X(86)
03 TXLLN   PIC 9(4) COMP  * LINE LENGTH READ
03 TXSIG   PIC 9(4) COMP  * SIGNIFICANT INDEX
03 FILLER  PIC X(4)
03 TXNCH   PIC 9(4) COMP  * CHARACTERS IN BODY
```

These three fields are established by a READ, READ FIRST or READ NEXT operation which is successful, or which suffers a record length exception.

TXLLN contains the total length of the line read. This excludes the terminating NUL character supplied by the access method but includes the new-line sequence that precedes the body of the line. If a record length exception occurs TXLLN will indicate the length of the line before it was truncated.

TXSIG indexes the first significant character of the line. If there is none TXSIG will index the NUL character supplied as a terminator.

TXNCH contains the number of characters in the body of the line. If a record length exception occurs TXNCH will indicate the length of the body before the line was truncated.
For example, suppose the record last returned by the access method was:

```
CR LF HT SP SP A B C NUL
```

and no record length exception was signalled. Then TXLLN would contain 8, TXSIG 6 (indexing the letter A) and TXNCH 6.

5.3 The OPEN Statement

The OPEN statement is coded:

```
OPEN type filename
```

where type is the word NEW, OLD or SHARED and filename identifies the text file definition. An OPEN statement must be executed before any other operation affecting the file. If an OPEN is attempted but the FD is already open your program will be terminated in error.

OPEN NEW is used to create a TF file. OPEN OLD obtains exclusive access to an existing file and OPEN SHARED allows co-operating jobs running under multi-user System Manager to share a direct access TF file. (The features of the open operations which are common to all file organisations, such as volume-id checking, are described in detail in section 1.3.2.)

5.3.1 File Conditions

When an OPEN NEW is used to access a direct access volume the System Manager will signal file already exists if a file with the same file-id as that specified in the FD is present.

When an OPEN OLD or OPEN SHARED statement is executed the System Manager checks to see whether a file with text file organisation and the same file-id as that specified by the FD is present on the volume. If this is not the case wrong type ($$RES = "1") or file not found ($$RES = "3") is signalled.

5.3.2 Successful Completion

Providing no file condition or irrecoverable I/O error occurs an OPEN NEW results in the System Manager allocating the amount of space indicated by the FD's SIZE statement (or its absence). For a direct access file the write next pointer will be set to address the first character so that a subsequent WRITE NEXT operation will create the first record.

When an OPEN OLD or OPEN SHARED statement completes successfully the file size is returned in the FD and can be accessed by the user program if a SIZE statement with the symbol option was coded.

5.4 The WRITE NEXT Statement

WRITE NEXT is used to create the records of a new TF file, or extend an existing direct access TF file by writing a new record at its end. It is coded:

```
WRITE NEXT filename FROM A
```

Here filename identifies the text file definition and A is a simple or indexed variable. If a WRITE NEXT is attempted on an FD which is not already open the program will be terminated in error.

5.4.1 File Conditions
The file space exhausted condition will be signalled if there is insufficient space remaining in the extent to write the new record.

5.4.2 Successful Completion
Providing no file condition or irrecoverable I/O error occurs WRITE NEXT will transfer characters byte by byte from A to the file, starting at the character addressed by the write next pointer. All characters up to and including the terminating NUL within A will be transferred. The write next pointer will be incremented to address the next record.

The starting character number of the line written will be returned in the keyname field and can be accessed by the user program if the KEY statement was coded in the file definition.

5.4.3 Programming Notes
The record written by a WRITE NEXT statement may contain a number of lines, separated by newline sequences. The record can be up to 32766 bytes in length, although individual lines within the record should be no longer than 256 bytes so that they can be accessed using READ or READ NEXT. If you are adding a number of lines to a text file it is advisable to combine them together into larger records, as this reduces the number of disk accesses required, and hence improves performance.

5.5 The READ FIRST Statement
READ FIRST is used to read the very first record in the file. It is coded :

```
READ FIRST filename INTO A
```

Here filename identifies the text file definition and A is a simple or indexed variable. If a READ FIRST is attempted on an FD which is not open the program will be terminated in error.

5.5.1 File Conditions
The end of file condition will be signalled if the file is empty (contains no records).

5.5.2 The Record Length Exception
Providing no file condition or irrecoverable I/O error occurs, READ FIRST sets the key to address the very first record on the file (a value of zero), and reads the line.

If the line is longer than 256 characters then the program will be terminated in error. If the length is less than 256, but greater than or equal to the record length specified in the FD, then although the line can be handled, it will be truncated before being supplied to your program. In this case the first (record length - 1) characters are transferred to A, followed by the NUL character, then exception 3 is signalled. The extra characters are ignored, and a subsequent READ NEXT will read the next line on the file.

5.5.3 Successful Completion
Providing no file condition, record length exception or irrecoverable I/O error occurs READ FIRST will transfer the first line of the file

to A, and then place a NUL character in A following the end of the line.

5.6 The READ NEXT Statement

READ NEXT is used to process part or all of a file sequentially. It is coded:

```
READ NEXT filename INTO A
```

Here filename identifies the text file definition and A is a simple or indexed variable. If a READ NEXT is attempted on an FD which is not open the program will be terminated in error.

5.6.1 File Conditions

The end of file condition will be signalled on attempting to read past the used part of the file. For a direct access file this is delimited by the write next pointer.

5.6.2 The Record Length Exception

Providing no file condition or irrecoverable I/O error occurs, READ NEXT increments the key by the length of the previous line accessed and then reads the line thus identified.

If the line read is longer than 256 characters then the program will normally be terminated in error. However, if the length is less than 256 but greater than or equal to the record length specified in the FD, then although the line can be handled, it will be truncated before being supplied to your program. In this case the first (record length - 1) characters are transferred to A, followed by a NUL character, then exception 3 is signalled. The extra characters are ignored, and a subsequent READ NEXT will read the next line on the file.

5.6.3 Successful Completion

Providing no file condition, record length exception, or irrecoverable I/O error occurs READ NEXT will transfer the line read to the input area A, and then place a NUL character in A following the end of the line.

5.6.4 Programming Notes

If a KEY statement was not specified in the file definition the FD still contains an internal key field which is set to 0 when the file is opened. In this case READ NEXT can be used to read sequentially through the entire file and READ rereads the last record thus retrieved.

When a KEY statement is specified a READ can be used to position the file at any point. READ NEXT will continue sequential processing from the record following the one retrieved by READ.

The WRITE NEXT statement updates the key field to address the last record of the file, so a READ NEXT following WRITE NEXT will be suppressed with an end of file condition.

Following a successful READ NEXT, or one which suffered a record length exception, three special fields are returned to the program in the FD. These indicate the length of the current line, the position of its first significant character, and the size of its body (the part following the newline sequence). For more details refer to 1.2.10.

5.7 The READ Statement

READ is used to retrieve a line at random or reread a record retrieved by READ NEXT. It is coded:

```
READ filename INTO A
```

Here filename identifies the text file definition and A is a simple or indexed variable. If READ is attempted on an FD which is not already open the program will be terminated in error.

5.7.1 Establishing the Key

If READ is to be used as a random access operation the FD must contain a KEY statement coded as:

```
KEY IS symbol
```

which causes:

```
02 symbol PIC 9(9) COMP
```

to be generated. The program must then move the character number of the start of the line it is required to retrieve to this field before executing the READ statement.

If a KEY statement is not coded in the file definition the FD still contains an internal key field which is set to zero when the file is opened and incremented by READ NEXT or set by WRITE NEXT. In this case the only use of the READ statement is to reread the record previously retrieved by READ NEXT or written by WRITE NEXT.

5.7.2 File Conditions

The file boundary violation condition will be signalled if the record number supplied in the keyname field lies outside the used part of the file delimited by the write next pointer.

5.7.3 The Record Length Exception

Providing that no file condition or irrecoverable I/O error occurs a line is read from the file. If the line is longer than 256 characters the program will normally be terminated in error. However, if the length is less than 256 but greater than or equal to the record length specified in FD, then although the line can be handled, it will be truncated before being supplied to your program. In this case the first (record length - 1) characters are transferred to A, followed by a NUL character, then exception 3 is signalled. The extra characters are ignored, and a subsequent READ NEXT will read the next line of the file.

5.7.4 Successful Completion

Providing no file condition, record length exception, or irrecoverable I/O error occurs then READ will transfer the line identified by the key from the file to area A, and then place a NUL character in A following the end of the line.

5.7.5 Programming Notes

Following a successful READ, or one which suffered a record length exception, three special fields are returned to the program in the FD.

These indicate the length of the current line, the position of its first significant character, and the size of its body (the part following the newline sequence).

5.8 The CLOSE Statement

CLOSE must be used to terminate the processing of a file. It is coded:

```
CLOSE filename [TRUNCATE|DELETE]
```

where filename identifies the text file definition.

5.8.1 Standard Processing

CLOSE always completes any outstanding I/O operations on the file and returns the FD to the status it possessed prior to being opened. Following CLOSE, the FD can be re-opened for the same, or a different, text file by a subsequent OPEN NEW or OPEN OLD statement.

5.8.2 File Conditions

If the file is not open the file not open condition will be signalled.

5.8.3 Truncation

The TRUNCATE phrase instructs the System Manager to return any spare space at the end of the extent to the volume so it can be re-allocated.

5.8.4 Deletion

The DELETE phrase causes all the space occupied by a direct access file to be returned to the volume and its file-id to be erased from the directory. Thus following a CLOSE DELETE the affected file no longer exists.

5.8.5 Programming Notes

If you fail to close a direct access file any new records created by WRITE NEXT since the time it was opened will be lost. This is because the updated write next pointer is only preserved on the volume by the CLOSE or CLOSE TRUNCATE operations. If you are using a WRITE NEXT to infrequently add records to the end of a text file during interactive working you should consider "check-pointing" the file by closing and re-opening it following each addition.

6. The Basic Direct File Organisation

6.1 Basic Direct Files

A BD file must always be assigned to a direct access device, and is treated as a string of bytes numbered sequentially from zero upwards. Records are accessed by supplying the starting byte number as key, together with the number of bytes to be read or written.

6.1.1 Record Format

The format of BD records is entirely under program control since you determine the length and starting byte of each record. This makes the basic direct file organisation particularly suitable for handling special file structures such as databases.

6.1.2 Specifying File Attributes

The attributes of a file, such as its unit-id, volume-id and file-id, are specified in its file definition (FD), coded in the data division. When creating a new file you specify the space to be allocated in the FD.

You must specify the name of the area which contains the record length. You can also define a key area to contain the starting byte number for use in random access READ and WRITE statements. Section 6.2 below describes those parts of the file definition which are specific to the basic direct file organisation, but the statements which are common to all organisations are defined in section 1.2.

6.1.3 File Processing Statements

Nine procedure division statements are provided to enable a BD file to be processed. They are:

OPEN which must be executed prior to any other statement affecting the file;

READ to read a record at random;

READ FIRST to read the very first record in the file;

READ LAST to read the very last record in the file;

READ NEXT to read the next record during sequential processing;

READ PRIOR to read the previous record during sequential processing;

WRITE to update an existing record at random;

WRITE NEXT to write a record during sequential processing;

CLOSE to terminate processing of a file.

When any of the statements is executed a file condition, signalled by exception condition 2, may arise as explained in section 1.3.1. In addition, if OPTION ERROR or ON ERROR is specified in the FD, an exception condition will be generated should an irrecoverable I/O error occur, as explained in section 1.6. Therefore it is usual to

follow each file processing statement with an ON EXCEPTION statement. If you do not, and an exception condition arises, your program will be terminated in error.

6.1.4 Extent Boundary Checking

A write next pointer is not maintained for a basic direct file, and you may use WRITE NEXT, WRITE, READ FIRST, READ LAST, READ NEXT, READ PRIOR and READ statements to access information anywhere within the allocated file extent. However, should any of these statements attempt to read or write a record which is partially or wholly outside the file extent, the file boundary violation file condition will be signalled.

6.1.5 The ORGANISATION Statement

If a program uses the basic direct access method then the statement:

```
ORGANISATION OR$85 TYPE x EXTENSION 8
```

must be coded in the data division before the first FD or data declaration. The type, x, may be any value in the range 0 to 99. It is the organisation type that will be given to any files created by OPEN NEW, and which will appear in a directory listing. We recommend that users avoid using organisations 0 to 9 as these are reserved for use by the System Manager. (For a more detailed explanation of the ORGANISATION statement, refer to section 1.2.)

6.2 The File Definition

The file definition for a basic direct file is coded in either working storage or the linkage section as follows:

```
FD filename ORGANISATION OR$85
[ASSIGN TO UNIT unit-id FILE file-id [VOLUME volume-id]]
[KEY IS keyname]
RECORD LENGTH IS length
[SIZE IS size]
[OPTION ERROR]
[ON ERROR intercept]
```

The FD establishes a special group data item, 88 bytes in length, whose name is filename. The quantities unit-id, file-id, volume-id, keyname, length and size appear as subordinate items within this group and can, if need be, be referred to by the application program.

If it is possible to specify unit-id, file-id or volume-id before the program executes then the quantity should be coded as a character string in quotes. If you can specify the size or length before the program executes, code the quantity as a numeric string. If any of these quantities is not known until run-time then a symbol must be coded for the quantity. This symbol will then label a level 02 item which the user program is responsible for initialising.

6.2.1 The Filename

The filename must be a symbol. It serves to label the file definition, as explained in section 1.2.1.

6.2.2 The ORGANISATION Clause

The ORGANISATION clause must be coded as shown. It indicates that the basic direct access method is to be used.

6.2.3 The ASSIGN Statement
Use of the ASSIGN statement, which is the same for any file organisation, is explained in section 1.2.

6.2.4 Optional Statement Placement
The KEY, RECORD LENGTH, SIZE, OPTION and ON ERROR statements may appear in any order following the ASSIGN statement. The RECORD LENGTH statement must be coded: the others are optional.

6.2.5 The KEY Statement
The KEY statement is only required if records are to be accessed at random. Keyname is specified as a symbol and the statement:

```
02 symbol PIC 9(9) COMP
```

is generated within the FD.

BDAM maintains the keyname field to contain the byte number of the start of the record last accessed. The byte number of the very first byte of the file counts as byte number zero.

The program must place the byte number of the start of the record it requires to access in the keyname field before executing a random READ or WRITE operation.

A successful READ NEXT or WRITE NEXT operation increments the keyname field by the length of the previous record accessed and then accesses the record thus identified: if the previous operation on the file was an OPEN then the first record is accessed.

A successful READ PRIOR decrements the keyname field by the length of the record to be read, and then accesses that record.

6.2.6 The RECORD LENGTH Statement
The RECORD LENGTH statement is always required. Length is specified as a symbol and the statement:

```
02 symbol PIC 9(4) COMP
```

is generated within the FD.

The program must place the length of the record to be read in the length field before executing a read or write statement. The field is set to zero when the file is opened, and is not altered by read and write operations.

6.2.7 The SIZE Statement
The SIZE statement is required when creating a new file or truncating an existing one. Its use, which is common to all file organisations, is explained in section 1.2.8.

6.2.8 The OPTION and ON ERROR Statements
OPTION ERROR should be coded only if you wish your program to regain control following an irrecoverable I/O error. ON ERROR should be coded if you wish to handle certain I/O errors specially. The processing of these statements is common to all file organisations, and is described in section 1.6.

6.2.9 Additional Fields in the FD

A basic direct access FD contains three additional fields which can be accessed by including a redefinition of the FD, as in the following example:

```
01    BD REDEFINES filename
03 FILLER PIC X(44)
03 BDLAB
05 FILLER PIC X(3)
05 BDORG PIC 9(2) COMP
05 FILLER PIC X(12)
05 BDADE PIC X(8)
05 FILLER PIC X(12)
```

The field BDLAB, the file label area, is a 36-byte area Containing all the information about the file which is preserved in its label, including the organisation type, file-id and size. It is written to the file label by a successful CLOSE, and is returned by a successful OPEN OLD or OPEN SHARED. A successful OPEN NEW statement may corrupt certain parts of the label area.

If you use the basic direct access method in conjunction with a link handler constructed using Assembler Interface to transmit files of any organisation across a communications link then, in addition to the data, the label area should also be transmitted.

You should then use the basic direct access method to write the transmitted data to the direct access device. The file label area, containing all the access method dependent information, can then be written to the file label by moving it to the output file label area in the BD FD before closing the file. In this way it is possible to transmit files of any organisation, including relative sequential, indexed sequential and program files. There is a detailed example illustrating the technique in Appendix D, which contains a short program capable of copying files of any organisation using the basic direct access method.

Following a successful OPEN OLD or SHARED operation the field BDORG will contain the organisation type of the file opened, as specified in the ORGANISATION statement associated with the FD. This can be checked by your program to ensure that the organisation is as expected.

The field BDADE, the access method dependent area, is an 8-byte area which is available to the user program. It is written to the file label by a successful CLOSE statement, and is returned by a successful OPEN OLD or SHARED. Following an OPEN NEW operation the area will be set to binary zeros.

6.3 The OPEN Statement

The OPEN statement is coded:

```
OPEN type filename
```

where type is the word NEW, OLD or SHARED and filename identifies the basic direct file definition. An OPEN statement must be executed before any other operation affecting the file. If an OPEN is attempted but the FD is already open your program will be terminated in error.

OPEN NEW is used to create a BD file. OPEN OLD obtains exclusive access to an existing file and OPEN SHARED allows co-operating jobs running under multi-user System Manager to share a file. (The features of the open operation which are common to all file organisations, such as volume-id checking, are described in detail in section 1.3.2.)

6.3.1 File Conditions

The file already exists condition will be signalled in response to OPEN NEW if a file with the same file-id as that specified in the FD is already present on the direct access volume.

When an OPEN OLD or OPEN SHARED statement is executed the System Manager checks to see whether a file with the same file-id as that specified in the FD is present on the volume, and that it is not a product file. If this is not the case, file not found is signalled. (Note that the organisation is not checked as is normally the case. If you wish to test that the organisation is as expected, examine BDORG as explained in 6.2.9.)

6.3.2 Successful Completion

Providing no file condition or irrecoverable I/O error occurs an OPEN NEW results in the System Manager allocating the amount of space indicated by the FD's SIZE statement (or its absence).

When an OPEN OLD or OPEN SHARED statement completes successfully the file size is returned in the FD and can be accessed by the user program if a SIZE statement with the symbol option was coded. In addition the file organisation type and access method dependent area are returned in BDORG and BDADE, as explained in 6.2.9.

The record length field in the FD is set to zero by a successful OPEN NEW statement. When an OPEN OLD or OPEN SHARED statement completes successfully the record length returned is the value it had when the file was last closed. If the file is an IS or RS file this will be the actual record length of the file.

The keyname field in the FD is always set to zero following a successful OPEN statement. This is to allow the file to be processed sequentially using a sequence of READ NEXT or WRITE NEXT statements as described below.

6.3.3 Programming Notes

Since the basic direct access method does not check the file organisation it can be used to access any file, except a product file such as the System Manager command library. You should not, however, attempt to update a file created by a different access method, as this is likely to result in the file becoming corrupt.

6.4 The WRITE NEXT Statement

WRITE NEXT is used to write all or part of a file sequentially. It is coded:

```
WRITE NEXT filename FROM A
```

Here filename identifies the basic direct file definition and A is a simple or indexed variable. If a WRITE NEXT is attempted on an FD which is not already open the program will be terminated in error.

6.4.1 Establishing the Length

The FD must contain the RECORD LENGTH statement with the length coded as a symbol. The program must establish the length of the record to be written (1 to 32767 bytes) in this field before executing the WRITE NEXT.

If the length is not established explicitly by the program the length of the last record accessed will be used.

6.4.2 File Conditions

An file boundary violation condition will be signalled if WRITE NEXT attempts to output a record which is wholly or partially outside the file extent.

6.4.3 Successful Completion

Provided that no file condition or irrecoverable I/O error occurs WRITE NEXT will set the key to the number of the byte following the previous record accessed (if any) and transfer A to the record thus identified. The number of bytes transferred is given by the length field.

6.4.4 Programming Note

If a KEY statement was not specified in the file definition the FD still contains an internal key field which is set to zero when the file is opened. In this case WRITE NEXT can be used to write the entire file sequentially.

When a KEY statement is specified a READ or WRITE can be used to position the file at any point. WRITE NEXT will continue writing from the record following the one accessed by READ or WRITE.

The READ NEXT statement sets the key field to the record retrieved, so that a subsequent WRITE NEXT statement will write the following record.

6.5 The WRITE Statement

WRITE is used to write a record at random or rewrite the last record accessed. It is coded:

```
WRITE filename FROM A
```

Here filename identifies the basic direct file definition and A is a simple or indexed variable. If a WRITE is attempted on an FD which is not already open the program will be terminated in error.

6.5.1 Establishing the Key

If WRITE is to be used as a random access operation the FD must contain a KEY statement coded as:

```
KEY IS symbol
```

which causes:

```
02 symbol PIC 9(9) COMP
```

to be generated. The program must then move the byte number of the start of the record it is required to retrieve to this field before executing the WRITE statement.

If a KEY statement is not coded in the file definition the FD still contains an internal key field which is set to zero when the file is opened and incremented by READ NEXT and WRITE NEXT. In this case the only use of the WRITE statement is to rewrite the record previously retrieved by READ NEXT or written by WRITE NEXT.

6.5.2 Establishing the Length

The FD must contain the RECORD LENGTH statement with the length coded as a symbol. The program must establish the length of the record to be written (1 to 32767 bytes) in this field before executing the WRITE.

If the length is not established explicitly by the program the length of the last record accessed will be used.

6.5.3 File Conditions

An extent boundary violation condition will be signalled if WRITE attempts to output a record which is wholly or partially outside the file extent.

6.5.4 Successful Completion

Provided that no file condition or irrecoverable I/O error occurs WRITE will transfer the number of bytes given by the record length from A to the record identified by the key.

6.5.5 Programming Note

A WRITE operation will always complete the updating of the file before returning control, so that even if the program subsequently fails the file will have been updated.

6.6 The READ FIRST and READ LAST Statements

READ FIRST is used to read the very first record of the file. It is coded:

```
READ FIRST filename INTO A
```

Similarly READ LAST is used to read the very last record in the file. It is coded:

```
READ LAST filename INTO A
```

In both cases filename identifies the basic direct file definition and A is a simple or indexed variable. If READ FIRST or READ LAST is attempted on an FD which is not open the program will be terminated in error.

6.6.1 Establishing the Length

The FD must contain the RECORD LENGTH statement with the length coded as a symbol. The program must establish the length of the record to be read (1 to 32767 bytes) in this field before executing a READ FIRST or READ LAST.

If the length is not established explicitly by the program the length of the last record accessed will be used.

6.6.2 File Conditions

A file boundary violation condition will be signalled if a READ FIRST or READ LAST attempts to input a record which is larger then the total file size, and which would therefore start or end outside the file extent.

6.6.3 Successful Completion

Provided that no file condition or irrecoverable I/O error occurs READ FIRST will set the key to address the first byte of the file (a value of zero), and READ LAST will set the key to be the file extent size less the length of the record to be processed. The record thus identified will be transferred to A, the number of bytes transferred being given by the length field.

6.6.4 Programming Note

Use of READ FIRST is normally used to reposition at the start of the file prior to reading records sequentially using READ NEXT.

Use of READ LAST presupposes that the program has sufficient information about the file to be able to determine the length of the very last record (possibly because the length is fixed). It is important to note that it is the responsibility of the program to determine an appropriate record length for use by READ LAST, not that of the basic direct access method itself.

6.7 The READ NEXT and READ PRIOR Statements

READ NEXT and READ PRIOR are used to process part or all of a file sequentially. READ NEXT is used to read the next sequential record, and it is coded:

```
READ NEXT filename INTO A
```

READ PRIOR is used to read the previous sequential record. It is coded:

```
READ PRIOR filename INTO A
```

In both cases filename identifies the basic direct file definition and A is a simple or indexed variable. If a READ NEXT or READ PRIOR is attempted on an FD which is not already open the program will be terminated in error.

6.7.1 Establishing the Length

The FD must contain the RECORD LENGTH statement with the length coded as a symbol. The program must establish the length of the record to be read (1 to 32767 bytes) in this field before executing the READ NEXT or READ PRIOR.

If the length is not established explicitly by the program the length of the last record accessed will be used.

6.7.2 File Conditions

A file boundary violation condition will be signalled if READ NEXT or READ PRIOR attempts to input a record which is wholly or partially outside the file extent.

6.7.3 Successful Completion

Provided that no file condition or irrecoverable I/O error occurs READ NEXT will set the key to the number of the byte following the previous record accessed (if any), and READ PRIOR will set the key to the number of the byte record length bytes before the start of the previous record accessed. The record thus identified is transferred to A. In both cases the number of bytes transferred is given by the length field.

6.7.4 Programming Note

If a KEY statement was not specified in the file definition the FD still contains an internal key field which is set to 0 when the file is opened. In this case READ NEXT can be used to read sequentially through the entire file; READ PRIOR at any point may be used to retrieve the preceding record; WRITE updates the last record thus retrieved; and READ rereads it. In addition READ FIRST and READ LAST may be used to position at either the start or end of the file.

When a KEY statement is specified a READ or WRITE can be used to position the file at any point. READ NEXT will continue sequential processing from the record following the one retrieved by READ or output by WRITE, and READ LAST will process sequentially from the record before it.

A WRITE NEXT statement sets the key field to the start of the record written, so that a subsequent READ NEXT statement will read the following record, and a READ PRIOR statement will read the previous record.

Note that when using READ PRIOR it is the responsibility of the program to determine the length of the record to be processed by some means (possibly because it has a fixed length), and not that of the basic direct access method itself.

6.8 The READ Statement

READ is used to retrieve a record at random or reread the last record accessed. It is coded:

```
READ filename INTO A
```

Here filename identifies the basic direct file definition and A is a simple or indexed variable. If READ is attempted on an FD which is not already open the program will be terminated in error.

6.8.1 Establishing the Key

If READ is to be used as a random access operation the FD must contain a KEY statement coded as:

```
KEY IS symbol
```

which causes:

```
02 symbol PIC 9(9) COMP
```

to be generated. The program must then move the byte number of the start of the record it is required to retrieve to this field before executing the READ statement.

If a KEY statement is not coded in the file definition the FD still contains an internal key field which is set to zero when the file is opened and incremented by READ NEXT and WRITE NEXT. In this case the only use of the READ statement is to reread the record previously retrieved by READ NEXT or written by WRITE NEXT.

6.8.2 Establishing the Length

The FD must contain the RECORD LENGTH statement with the length coded as a symbol. The program must establish the length of the record to be read (1 to 32767 bytes) in this field before executing the READ.

If the length is not established explicitly by the program the length of the last record accessed will be used.

6.8.3 File Conditions

A file boundary violation condition will be signalled if READ attempts to input a record which is wholly or partially outside the file extent.

6.8.4 Successful Completion

Providing no file condition or irrecoverable I/O error occurs, READ will transfer to A the number of bytes given by the record length of the record specified in the key field.

6.8.5 The READ PHYSICAL Statement

As the key of a BDAM file is a byte number, the READ PHYSICAL statement functions in exactly the same way as a READ.

6.9 The CLOSE Statement

CLOSE must be used to terminate the processing of a file. It is coded:

```
CLOSE filename [TRUNCATE|DELETE]
```

where filename identifies the basic direct file definition.

6.9.1 Standard Processing

CLOSE always completes any outstanding I/O operations on the file and returns the FD to the status it possessed prior to being opened. Following CLOSE, the FD can be re-opened for the same (or a different) file by a subsequent OPEN NEW or OPEN OLD statement.

6.9.2 File Conditions

If the FD passed to the CLOSE was not open, then a file not open condition will be signalled.

6.9.3 Truncation

The TRUNCATE phrase is used in conjunction with a SIZE statement with the symbol option. The program must have established the size to which the file is to be truncated in the size field. When a CLOSE TRUNCATE is subsequently executed the additional part of the file, if any, will be returned to the volume so that it can be reallocated.

If you have created a new file using a sequence of WRITE NEXT Statements and wish to return any unused space to the System Manager you must have specified the key field using the symbol option. To find the size of the file you have created simply add the record length of the last record output by WRITE NEXT and the current value of the

keyname field together. Store this value in the size field, then CLOSE TRUNCATE the file.

Note that if a CLOSE TRUNCATE is attempted and the size specified is greater than the file's extent, the program will be terminated in error.

If you use the basic direct access method on a file created by a different access method, you must never truncate the file under any circumstances, except to set it to the current file size, as given by the FSTAT$ system routine.

6.9.4 Deletion

If a DELETE phrase is coded all the space the file occupies is returned to the volume and its file-id is destroyed. Following a CLOSE DELETE the file no longer exists.

6.9.5 Programming Notes

If you fail to close a basic direct file, records created by WRITE NEXT statements may not have been written to the file, as such records are buffered for efficiency.

A WRITE operation is effected immediately so that a file which is only updated by WRITEs should normally remain consistent, even if it is not closed. This feature is intended to protect files which are updated interactively from damage in the event of machine failure: programs should always close the files they use.

A CLOSE statement will write the access method dependent area in the FD (defined in 6.2.9) to the file label, so that it can be retrieved by a subsequent OPEN OLD or OPEN SHARED.

7. The Data Library File Organisation

7.1 Data Library Files

A data library file contains up to 99 fixed length records, identified by 10 character names. Each block contains a title, date and time of creation, and the operator-id of the creator, as well as user-defined fields. In addition to allowing records to be read or written by name, the access method will list out the library members on the screen, so that you can identify the one you want and delete members that are no longer needed.

This access method should be used when you need to store a number of fixed length items of information with user-defined names, but do not want to create large numbers of separate files, nor have to list existing items or delete unwanted items yourself.

In addition to the data records, there is an extra header record, containing the library title. This can be used to hold additional information specific to the data library.

It is not necessary to allocate space for all 99 members: if the library is smaller it is simply restricted to the number of members for which there is space. You can increase its size, up to a maximum of 99 members, by simply copying it to a larger file.

7.1.1 File Format

A data library file consists of a 200-byte index block followed by up to 100 records, each of the length given by the file's record length. The first of these records is the header record, the rest are data records 1 to 99.

The first 2 bytes of the index are not used. The remainder consist of a 1 byte hash of the record name, followed by the number of the corresponding record, with the hash set to zero if the record does not exist. The entries are held with all the in use records before all the unused entries, and with in use records held in alphabetic order.

The hash of the record name is formed by adding together all the characters of the name and taking the junior byte of the result, but with a hash value of zero being replaced by 1 so that the value zero is available for unused records.

When a new record is created the record number is taken from the first unused index entry; when a record is deleted its record number is put at the start of the unused record entries. In this way records near the start of the file are used in preference to later records.

7.1.2 Record Format

Each record starts with a 60 byte fixed format area which contains the record name, title and the operator-id, and the date and time of the last update. The remainder of the record is user defined. The layout of the first 60 bytes is as follows:

```
03 &&NAME PIC X(10)      * record name
03 &&TTL  PIC X(30)      * title
03 &&OPID PIC X(4)       * operator
03 &&DATE PIC 9(6) COMP  * date yymmdd
03 &&TIME PIC X(8)       * time hh:mm:ss
```

```
     03 FILLER PIC X(5)       * reserved for future use
```

7.1.3 File Processing Statements

Ten procedure division statements are provided:

OPEN which must be executed prior to any other statement affecting the file;

READ to read a record given the member name or list the library;

READ FIRST to read the first record in the library;

READ LAST to read the last record in the library;

READ NEXT used to read records sequentially in ascending order;

READ PRIOR to read records sequentially in descending order;

WRITE used to add a new record or update an existing record with the name specified;

REWRITE used to update the record last read, possibly changing its name or deleting it;

DELETE to delete a record from the library;

CLOSE to terminate processing of a library.

The READ and WRITE statements also allow you to list out the current contents of the library, and to delete unwanted records, as described below.

When any of the statements is executed a file condition, signalled by exception condition 2, may arise as explained in section 1.3.1. In addition, if OPTION ERROR or ON ERROR is specified in the FD, an exception condition will be generated should an irrecoverable I/O error occur, as explained in section 1.6. Therefore it is usual to follow each file processing statement with an ON EXCEPTION statement. If you do not, and an exception condition arises, your program will be terminated in error.

7.1.4 Listing Format

When the library is listed it is displayed on the screen in the following format:

```
     library-title

     NAME TITLE      OPERATOR LAST UPDATED
     1 name   title      operator-id dd/mm/yy hh:mm
     2 "   "    "    "    "
     [*** end ***]

     Key name, number, Next page, Delete, End:
```

If you key a number, the corresponding record is selected. Replies of N or <CR> cause the next page to be displayed. If there are no more

entries, the list starts again at the beginning. A reply of D causes you to be prompted for the name of the entry to be deleted, and the screen is redisplayed. A reply of <ESCAPE> when $$ESC is set non-zero is treated the same as E, and causes an exit.

7.1.5 The ORGANIZATION Statement

If a program accesses a data library, the following statement must be coded in the data division before the first FD or data declaration:

```
ORGANIZATION OR$86 TYPE 5 EXTENSION 8
```

7.2 The File Definition

The file definition for a data library is coded as follows:

```
FD file-name ORGANIZATION OR$86
[ASSIGN TO UNIT unit-id FILE file-id [VOLUME volume-id]]
[RECORD LENGTH IS length]
[SIZE IS size]
[OPTION ERROR]
[ON ERROR intercept]
```

7.2.1 The Filename

The filename must be a symbol. It serves to label the file definition, as explained in section 1.2.1.

7.2.2 The ORGANISATION Clause

The ORGANISATION clause must be coded as shown. It indicates that the data library access method is to be used.

7.2.3 The ASSIGN Statement

Use of the ASSIGN statement, which is the same for any file organisation, is explained in section 1.2.

7.2.4 Optional Statement Placement

The RECORD LENGTH, SIZE, OPTION and ON ERROR statements may appear in any order following the ASSIGN statement. They are all optional.

7.2.5 The RECORD LENGTH Statement

The RECORD LENGTH statement is required when creating a new file, and may be present (in the symbol form, as described in section 1.2) when processing an existing file.

The record length of the library records must be established before an OPEN NEW takes place, and is returned whenever an OPEN OLD or OPEN SHARED is performed.

In addition whenever any READ, READ FIRST, READ LAST, READ NEXT, READ PRIOR, WRITE or REWRITE statement is performed, the amount of data processed for the record in question is determined by the current value of the record length, which may be any value from 60 up to the maximum record length.

7.2.6 The SIZE Statement

The SIZE statement is required when creating a new file. Its use, which is common to all file organisations, is explained in section 1.2.8.

7.2.7 The OPTION and ON ERROR Statements

OPTION ERROR should be coded only if you wish your program to regain control following an irrecoverable I/O error. ON ERROR should be coded if you wish to handle certain I/O errors specially. The processing of these statements is common to all file organisations, and is described in section 1.6.

7.3 The OPEN Statement

The open statement is coded:

```
OPEN type filename
```

where type is OLD or SHARED to access an existing file, or NEW to create a new file. The FD must be closed and the file-id and unit-id must have been established before the statement is executed; for an OPEN NEW the record length must also be established. The record length must be at least 60, the length of the fixed part of data records. The size is omitted or set to zero to allocate the maximum available space (subject to a maximum of 99 records), or set to the amount of space required, calculated as (200+ record length * (number of records + 1)).

If the size given in the FD is not at least (200 + record length) bytes (i.e. index plus header record), your program will be terminated in error.

7.3.1 File Conditions

A file not found ($$RES = "3") or wrong type ($$RES = "1") exception is signalled if the file is not present or not the correct organization for an OPEN OLD or OPEN SHARED operation.

A file already exists ($$RES = "4") exception is signalled if the file is already present when an OPEN NEW is executed.

An insufficient space ($$RES = "5") exception is signalled if the file is opened with size 0 and there is insufficient space on the volume for a file of minimum size (i.e. 200 + record length bytes).

7.3.2 Successful Completion

When the open completes successfully the record length and file size will be returned in the FD.

7.4 The WRITE Statement

WRITE is used to create or rewrite the record identified by the name field in the supplied record area. It is coded:

```
WRITE filename FROM A
```

Here filename identifies the data library file definition, and A may be a simple or indexed variable. If a WRITE is attempted on an FD which is not open then the program will be terminated in error.

7.4.1 The Record Length

The length of the record to be written must be established in the record length field. This may be shorter than the actual record length if you do not want to update the whole of the record.

7.4.2 The Record Name
You must establish the name of the record to be written in the first 10 bytes of the record area before executing the WRITE statement. If you supply the special record name of spaces then instead of writing a record, the library contents will be listed on the screen, and you will be allowed to delete unwanted entries.

7.4.3 File Full Processing
If the data library is full, then the current contents will be listed on the screen, and the user invited to delete one of the existing entries to make space:

```
.....
..... listing
.....

FILE FULL - Key Delete, Next page, End:
```

The processing is similar to the normal listing, except that if you delete a record, the new record overwrites it, and the operation completes successfully.

7.4.4 File Conditions
If the user abandons the write, without causing the new record to be written to the library, then a file space exhausted condition will be signalled.

7.4.5 Programming Notes
If the member name supplied is low-values then the header record will be written.

The current operator-id, date and time fields will be set up in the record when it is written.

Since the record name and title will be displayed on the screen when the record is listed, these fields should only contain characters in the range #20 to #7E.

7.5 The REWRITE Statement
REWRITE updates the data record last read or written, or the header record if the previous operation was an OPEN. It is coded:

```
REWRITE filename FROM A
```

Here filename identifies the data library file definition, and A may be a simple or indexed variable. If a REWRITE is attempted on an FD which is not open then the program will be terminated in error.

7.5.1 The Record Length
The length of the record to be written must be established in the record length field. This may be shorter than the actual record length if you do not want to update the whole of the record.

7.5.2 The Record Name
The first ten bytes of the record area supplied must contain the new record name. If a record with the new name already exists (other than

the record being updated) then the user will be asked to confirm that the existing record is to be overwritten:

```
OVERWRITE EXISTING RECORD name?
```

If you reply Y the existing record will be deleted, and the current record updated to have this name.

If the name supplied is spaces, the library will be listed instead.

7.5.3 File Conditions

If the user abandons the rewrite because the record with the new name already exists, then an already exists exception will be signalled.

7.5.4 Programming Notes

It is not possible to rewrite the header record with a name other than low-values, or to change any other record to have a name of low-values. Any such attempt will cause the program to be terminated in error.

The current operator-id, date and time fields will be set up in the record when it is written.

Since the record name and title are displayed on the screen when the record is listed, these fields should only contain characters in the range #20 to #7E.

7.5.5 Record Deletion

For compatibility with earlier version of the data library access method, if the record name area contains high-values when a REWRITE is performed, then the record will be deleted. In general, however, we would recommend that the DELETE statement, documented in section 7.6, is used to DELETE records from a data library.

7.6 The DELETE Statement

DELETE is used to remove the last record read or written from the data library. It is coded:

```
DELETE filename
```

Here filename identifies the data library file definition. If a DELETE is attempted on an FD which is not open then the program will be terminated in error.

Note that no record area need be passed as part of the DELETE statement, as the access method keeps a note of the last record accessed.

7.6.1 Programming Notes

A DELETE issued on a data library will remove from the index the record last accessed. In an environment where other programs may be sharing access to the data library it is sensible to use the LOCK statement to secure access to the record before it is deleted.

7.7 The READ Statement

READ is used to retrieve a record given its name, or to list out the records so that the user can identify the one required. It can also be used to allow unwanted records to be deleted. It is coded:

```
READ filename INTO A
```

Here filename identifies the data library file definition, and A may be a simple or indexed variable. If a READ is attempted on an FD which is not open then the program will be terminated in error.

7.7.1 The Record Length

The length of the record to be read must be established in the record length field. This may be shorter than the actual record length if you do not want to process the whole of the record.

7.7.2 The Record Name

The record supplied must have the record name established (in the first ten bytes) when the READ is issued. A name of low-values causes the header record to be retrieved; any other name except spaces causes the record of that name to be retrieved, or an exception to be signalled if it does not exist.

A name of spaces causes the library to be listed on the screen, and also allows the user to delete unwanted records.

7.7.3 File Conditions

If the specified record is not found, or you key <ESC> to the select prompt, a record not found exception will be signalled.

7.8 The READ FIRST and READ LAST Statements

READ FIRST is used to read the very first record in the index. It is coded:

```
READ FIRST filename INTO A
```

Similarly, READ LAST is used to read the very last record in the index. It is coded:

```
READ LAST filename INTO A
```

In both cases filename identifies a data library file definition, and A may be a simple or indexed variable. If a READ FIRST or READ LAST is attempted on an FD which is not already open the program will be terminated in error.

7.8.1 The Record Length

The length of the record to be read must be established in the record length field. This may be shorter than the actual record length if you do not want to process the whole of the record.

7.8.2 File Conditions

The record not found condition will be signalled to READ FIRST and READ LAST if there are no records present in the data library.

7.9 The READ NEXT and READ PRIOR Statements

READ NEXT is used to read each data record in turn from the data library, in ascending order of record name. It is coded:

```
READ NEXT filename INTO A
```

READ PRIOR is used to read sequential data records from the data library, in descending order of record name. It is coded:

```
READ PRIOR filename INTO A
```

In both cases filename identifies a data library file definition, and A may be a simple or indexed variable. If a READ NEXT or READ PRIOR is attempted on an FD which is not already open the program will be terminated in error.

7.9.1 The Record Length

The length of the record to be read must be established in the record length field. This may be shorter than the actual record length if you do not want to process the whole of the record.

7.9.2 The Record Retrieved

The record retrieved depends on the previous file operation.

For READ NEXT:

- following an OPEN the header record is read;
- following a read or write of the header record the first data record is read;
- following any other operation the next data record is read, or an exception signalled if there are no more.

For READ PRIOR:

- following an OPEN, or a read or write of the header record, the start of file condition will be signalled;
- following any other operation the previous data record is read, or an exception signalled if the start of file has been reached.

7.9.3 File Conditions

READ NEXT will signal the end of file condition when an attempt is made to read past the last non-deleted record in the library.

READ PRIOR will signal the start of file condition when an attempt is made to read before the first record in the library.

7.10 The CLOSE Statement

CLOSE must be used to terminate the processing of a file. It is coded:

```
CLOSE filename [TRUNCATE/DELETE]
```

where filename identifies the data library file definition.

7.10.1 Standard Processing

CLOSE always completes any outstanding I/O operations on the file and returns the FD to the status it possessed prior to being opened.

Following CLOSE, the FD can be re-opened for the same, or a different, file by a subsequent OPEN statement.

7.10.2 File Conditions

If a CLOSE is attempted on an FD which is not open, then the file not open condition will be signalled.

7.10.3 Truncation

If the TRUNCATE phrase is coded, the System Manager will return the unused part of the overflow area, if any, to the volume so that it can be re-allocated.

7.10.4 Deletion

If the DELETE phrase is coded all the space the file occupies is returned to the volume and its file-id is erased from the directory. Following a CLOSE DELETE the file no longer exists.

8. The Physical Sector Access Method

The physical sector access method allows you to read or write any sector of certain types of direct access volume according to the physical sector number which you supply. The access method will operate on discrete direct access volumes or entire domains, but it cannot be used on a subunit of a domain. It enables you to process native volumes which have not been created under the System Manager.

8.1 Specifying Volume Attributes

You use a standard Global Cobol file definition (FD) to specify the unit-id of the volume you require to access and, optionally, fields in which its physical sector size and total capacity will be returned. You also define a key field which you must set to contain the physical sector number before performing a READ or WRITE operation. Section 8.2 below explains those parts of the file definition which are specific to the physical sector file organisation, but the statements which are common to all organisations are defined in section 1.2 of this manual.

8.1.1 Processing Statements

Just four procedure division statements are provided to enable you to process the physical sectors of a volume. These are:

```
OPEN OLD        executed prior to accessing any sector;

READ            to read an identified sector;

WRITE                to write an identified sector;

CLOSE           to terminate processing of the volume.
```

When any of the statements is executed a file condition, signalled by exception condition 2, may arise as explained in section 1.3.1. In addition, if OPTION ERROR is specified in the FD, exception condition 1 will be generated should an irrecoverable I/O error occur. Therefore it is usual to follow each file processing statement with an ON EXCEPTION statement. If you do not, and an exception condition arises, your program will be terminated in error.

8.1.2 The ORGANISATION Statement

If a program uses the physical sector access method then the statement:

```
ORGANISATION OR$82 TYPE 2 EXTENSION 8
```

must be coded in the data division before the first FD or data declaration.

8.2 The File Definition

The file definition for physical sector access is coded in either working storage or the linkage section as follows:

```
FD filename ORGANISATION OR$82
[ASSIGN TO UNIT unit-id FILE "?"]
KEY IS keyname
[RECORD LENGTH IS length]
[SIZE IS size]
```

```
[OPTION ERROR]
[ON ERROR intercept]
```

The FD establishes a special group data item, 88 bytes in length, whose name is filename. The quantities unit-id, keyname, length, and size appear as subordinate items within this group and can be referred to by the application program.

If it is possible to specify the unit-id before the program executes, then code it as a character string in quotes. The sector length and size of the volume in bytes are returned in the length and size fields respectively when an OPEN OLD operation completes. The length and size should therefore be coded as symbols if the RECORD LENGTH and SIZE statements are used.

8.2.1 The Filename

The filename must be a symbol. It serves to label the file definition, as explained in section 1.2.1.

8.2.2 The ORGANISATION Clause

The ORGANISATION clause must be coded as shown. It indicates that the physical sector access method is to be used.

8.2.3 The ASSIGN statement

The ASSIGN statement indicates the unit on which the volume which is to be accessed by physical sector resides. The FILE clause must be coded as shown. (The file information specified merely satisfies the compiler's syntax checking: file information is not used by the access method since the volume does not contain a System Manager file directory.)

The ASSIGN statement may be omitted if the file definition appears in the linkage section.

8.2.4 Optional Statement Placement

The KEY, RECORD LENGTH, SIZE and OPTION statements may appear in any order following the ASSIGN statement. The KEY statement must be coded: the others are optional.

8.2.5 The KEY Statement

The KEY statement defines a symbol, keyname, which causes the statement:

```
02 keyname PIC 9(9) COMP
```

to be generated within the FD. You must move the physical sector number of the sector you require to access to the keyname field before executing a READ or WRITE operation. Physical sector numbers are allocated consecutively starting from 1.

8.2.6 The RECORD LENGTH Statement

You may optionally code the RECORD LENGTH statement to define a field, length, in which the physical sector size, in bytes, is returned. The statement:

```
02 length PIC 9(4) COMP
```

is generated within the FD. The sector size is placed in this field as the result of a successful OPEN OLD statement.

8.2.7 The SIZE statement

The SIZE statement is optional. When present it defines a symbol, size, which causes the statement:

```
     02 size PIC 9(9) COMP
```

to be generated within the FD. The System Manager will return the size of the volume in this field when it is successfully opened. (Note that the volume size is the product of the maximum sector number and the sector size.)

8.2.8 The OPTION and ON Statements

OPTION ERROR should be coded only if you wish your program to regain control following an irrecoverable I/O error. ON ERROR should be coded if you wish to handle certain I/O errors specially. The processing of these statements is common to all file organisations and is described in section 1.6 of this manual.

8.2.9 Additional Fields in the FD

A physical sector FD contains two additional fields which can be accessed by including a redefinition of the FD, as in the following example:

```
     01 PS REDEFINES filename
      02 FILLER          PIC X(56)
      02 PSTRACK         PIC 9(2) COMP  * TRACKS/CYLINDER
      02 PSSECT          PIC 9(2) COMP  * SECTORS/TRACK
```

These fields, which are supplied by the access method once the FD has been satisfactorily opened, can be used in determining all the parameters associated with a conventional direct access volume, as summarised in Table 8.2 overleaf.

PARAMETER	METHOD OF CALCULATION
Volume capacity in bytes	From the size field named in the FD's SIZE statement
Sector size in bytes	From the length field named in the FD's RECORD LENGTH STATEMENT
Number of cylinders per volume	SIZE / (PSTRACK x PSSECT x length)
Cylinder size in bytes	PSTRACK x PSSECT x length

Number of tracks per cylinder	PSTRACK
Track size in bytes	PSSECT x length
Number of sectors per track	PSSECT
Number of sectors per cylinder	PSTRACK x PSSECT
Number of sectors per volume	SIZE / length
Number of tracks per volume	SIZE / (PSSECT x length)

Table 8.2 - Direct Access Volume Parameters and How They are Calculated

8.3 The OPEN OLD statement

OPEN OLD is used to begin the processing of the physical sectors of a volume. It is coded:

```
OPEN OLD filename
```

where filename identifies the physical sector file definition. OPEN OLD must be coded before any READ, WRITE or CLOSE statement affecting the volume.

If OPEN OLD is attempted and the FD is already open the program will be terminated with an error.

8.3.1 File Conditions

The invalid device type file condition will be signalled in response to OPEN OLD if the System Manager detects that the operation is being attempted on a unit which is not a discrete direct access device or a domain.

8.3.2 Successful Completion

Providing no file condition or irrecoverable I/O error occurs, the OPEN OLD operation completes successfully. The sector length and volume size will be returned in the symbols associated with the RECORD LENGTH and SIZE statements if these have been coded.

8.4 The READ Statement

READ is used to retrieve a single specified physical sector:

```
READ filename INTO A
```

Here filename identifies the physical sector file definition and A is a simple or indexed variable. If READ is attempted on an FD which is not already open the program will be terminated in error.

8.4.1 Identifying the Sector to be Read

The FD must contain a KEY statement of the form:

```
KEY IS symbol
```

which causes:

```
02 symbol PIC 9(9) COMP
```

to be generated internally. The program must move the physical sector number of the sector to be read to the symbol field before executing the READ operation.

8.4.2 File Conditions

The file boundary violation condition will be signalled if the physical sector number supplied using the KEY symbol does not correspond to a sector the volume contains. The number must be positive since the sectors are allocated numbers consecutively, starting at 1. The out of range condition can be trapped and processed by an ON EXCEPTION statement following the READ. If such an exception occurs the record area, A, will remain undisturbed.

8.4.3 Successful Completion

Providing no file condition or irrecoverable I/O error occurs, the READ will transfer the contents of the single specified sector to A.

8.5 The WRITE Statement

WRITE is used to output a single specified physical sector:

```
WRITE filename FROM A
```

Here filename identifies the physical sector file definition and A is a simple or indexed variable. If WRITE is attempted on a FD which is not already open the program will be terminated in error.

8.5.1 Identifying the Sector to be Written

The FD must contain a KEY statement of the form:

```
KEY IS symbol
```

which causes:

```
02 symbol PIC 9(9) COMP
```

to be generated internally. The program must move the physical sector number of the sector to be written to the symbol field before executing the WRITE operation.

8.5.2 File Conditions

The file boundary violation condition will be signalled if the physical sector number supplied using the KEY symbol does not correspond to a sector the volume contains. The number must be positive since the sectors are allocated numbers consecutively, starting at 1. The out of range condition can be trapped and processed by an ON EXCEPTION statement following the WRITE. If such an exception

occurs no output operation will take place and the data on the volume will remain undisturbed.

8.5.3 Successful Completion

Providing no file condition or irrecoverable I/O error occurs, the WRITE will transfer the contents of a single specified sector beginning at A to the volume.

8.6 The CLOSE Statement

CLOSE must be used to terminate the processing of a volume. It is coded:

```
CLOSE filename
```

where filename identifies the physical sector file definition.

8.6.1 Standard Processing

CLOSE returns the FD to the status it possessed prior to being opened. Following CLOSE the FD can be reopened for the same, or a different, volume by a subsequent OPEN OLD statement.

8.6.2 File Conditions

If a CLOSE is attempted on an FD which is not open, then a file not open condition will be signalled.

8.6.3 Programming Notice

All READ and WRITE operations using the physical sector access method are effected immediately, so the CLOSE operation itself performs no I/O on the volume. Nevertheless the FD should be closed once volume processing is finished to release the System Manager resources involved.

9. The Speedbase File Organisation

9.1 Speedbase Files

A Global Speedbase file consists of one or more record sets each of which can be accessed by one or more indexes. A full description of Speedbase files can be found in the Speedbase Development System manual. The file processing statements OPEN, READ, READ PRIOR, READ NEXT, READ FIRST, READ LAST, READ PHYSICAL, CLOSE and UNLOCK are used in conjunction with a file definition with ORGANISATION OR$99R to process an existing Speedbase file:

OPEN	must be executed prior to any other statement affecting the file;
READ	retrieves a specific record corresponding to a specified index key value;
READ NEXT	retrieves the next record in the sequence of the specified index;
READ PRIOR	retrieves the preceding record in the sequence of the specified index;
READ FIRST	retrieves the first record in the sequence of the specified index;
READ LAST	retrieves the last record in the sequence of the specified index;
READ PHYSICAL	retrieves a record using its relative record number (RNN);
UNLOCK	is used to release a lock obtained by a previous READ operation;
CLOSE	must be issued to terminate file processing.

Speedbase files can only be created and maintained on direct access devices.

When any of the statements is executed a file condition, signalled by an exception condition 2, may arise as explained in 1.2.1. In addition, if OPTION ERROR is specified in the FD, exception condition 1 will be generated should an irrecoverable I/O error occur. Therefore it as usual to follow each file processing statement with an ON EXCEPTION statement. If you do not, and an exception condition arises, your program will be terminated in error.

9.1.1 Data Record Format

The data record format of a Speedbase file is described in full detail in the Speedbase Manual. The complete record layout, including the system area, must be used for reading records using SPAM.

9.1.2 Programming Notes

The Speedbase access method described in this chapter is a read only access method allowing only the reading (not writing) of Speedbase records. Speedbase files can only be created and amended using the Speedbase facilities described in the Speedbase Development System

Manual. The detailed structure and limitations of the database is also described in the Speedbase Development manual.

9.1.3 Performance Guidlines for Global Speedbase Databases

READ, READ FIRST and READ LAST need to access one index block from each index level then at least one data record.

READ NEXT and READ PRIOR retrieve a single data record using information saved in the access method or in a save area supplied in an OPEN statement, and may need to read a single index block as well. If there is no information saved (which can occur after an OPEN statement or after a SPAM access on a separate FD) then READ NEXT and READ PRIOR require the same processing as a READ.

READ PHYSICAL retrieves a single data record from the file address supplied, but the relative record number must have been determined from another database operation previously.

READ PHYSICAL is the most efficient read operation, then READ NEXT and READ PRIOR, then READ, READ FIRST and READ LAST.

9.1.4 File Locking

The SPAM database access method provides two forms of locking. The locks are specified by appending the lock-type (LOCK or PROTECT) to any of the READ, READ NEXT, READ PRIOR, READ FIRST, READ LAST, READ PHYSICAL statements. The effect of the various locking options is as follows:

- LOCK obtains an exclusive lock on the record (region = record address) This is intended to serve the purpose of an update lock, held on a record while it is in the process of being updated. This record lock should not be needed in the read only context of this access method but is included here for completeness. This lock is equivalent to the Speedbase default lock.

- PROTECT obtains a shared lock on the record. This is intended to protect the record locked against deletion or some other major update. This lock is equivalent to the Speedbase protect lock

Any further READ type operation on the FD will release any outstanding locks, and will of course obtain the new lock option if such is specified. An UNLOCK or CLOSE will release any outstanding locks on the FD. The locking operations are compatible with the default locking and PROTECT lock in Speedbase and can be used in conjunction with Speedbase.

9.1.5 The ORGANISATION Statement for Global Speedbase Databases

The following statement must be coded in the data division before the first FD or data declaration:

```
ORGANISATION OR$99R TYPE 99 EXTENSION 144
```

9.2 The File Definition

The file definition of a Speedbase file is coded in either the working storage or linkage section as follows:

```
FD filename ORGANISATION OR$99R
```

```
[ASSIGN TO UNIT unit-id FILE file-id [VOLUME volume-id]
[RECORD LENGTH IS length]
[SIZE IS size]
[OPTION ERROR]
[ON ERROR intercept]
[01 FILLER REDEFINES filename
03 FILLER PIC X(81) * reserved
03 SPINDEX     PIC X(6) * index name
03 SPRNN  PIC 9(6) COMP * relative record number]
```

The FD establishes a special group data item, 220 bytes in length whose name is filename. The quantities unit-id, file-id, volume-id, SPINDEX, SPRNN, length, size and intercept appear as subordinate items within this and can, if need be, be referred to by the application program.

If it is possible to specify unit-id, file-id, volume-id, or index-name before the program executes then the quantity should be coded as a character string in quotes. If any of these quantities apart from index-name are not known until run-time then a symbol must be coded for the quantity. The symbol will then label a 02 level item which the user program is responsible for initialising.

9.2.1 The Filename

The filename must be a symbol. It serves to label the file definition, as explained in section 1 of the Global Cobol File Management Manual. The filename for the Speedbase file must not contain the "DB" prefix.

9.2.2 The ORGANISATION Clause

The organisation clause must be coded as shown, except that you may use the abbreviation ORG instead of ORGANISATION. It indicates that the file is a Speedbase database.

9.2.3 The ASSIGN Statement

Use of the ASSIGN statement, which is the same for any file organisation is explained in section 1.

9.2.4 Optional Statement Placements

The RECORD LENGTH, SIZE, ON ERROR and option statements are optional and may appear in any order following the ASSIGN statement. The index field in also optional.

9.2.5 The Index Name Redefinition

The index name (SPINDEX) is required to identify the index used to read the Speedbase file. The program must place the index name to be used in the field before the file is opened.

9.2.6 The Relative Record Number Redefinition

The relative record number (SPRNN) is only required if you wish to use the READ PHYSICAL operation on the database. After any successful file processing statement (other than OPEN, CLOSE or UNLOCK) the relative record number of the record last processed will be returned to you in the SPRNN field. You establish the correct record address in the SPRNN field before executing a READ PHYSICAL operation.

9.2.7 The RECORD LENGTH statement

The RECORD LENGTH statement need only be supplied if you require to determine the record length of the record set at run-time. You code:

```
RECORD LENGTH IS symbol
```

causing the statement:

```
02 symbol PIC 9(4) COMP
```

to be generated either the file definition. When the OPEN operation terminates successfully the record length will be returned to you in the field named symbol.

9.2.8 The SIZE Statement

The SIZE statement is required only when you wish to determine the actual number of bytes allocated to the database file. The size is coded as a symbol, causing the statement:

```
02 symbol PIC 9(9) COMP
```

to be generated within the FD. When the OPEN statement completes satisfactorily the actual number of bytes allocated to the file will be returned in the generated field.

9.2.9 The OPTION ERROR and ON ERROR Statements

OPTION ERROR should be code only if you wish your program to regain control following an irrecoverable I/O error. ON ERROR should be coded if you wish to handle certain I/O errors specially. The processing of these statements is common to all file organisations, and is described in section 1.

9.3 The OPEN statement

The OPEN statement is coded:

```
OPEN type filename
```

where type is the word OLD or SHARED and filename identifies the Speedbase file definition.

An OPEN statement must be executed before any other file processing statement. If an OPEN is attempted but the FD is already open your program will terminate in error.

OPEN NEW is not supported because Speedbase files are always created using Speedbase utilities as documented in the Speedbase Development System Manual. OPEN OLD obtains exclusive access to the file (allowing only one index to be used at a time) whilst the OPEN SHARED allows co-operating jobs to share a Speedbase database file. The features of the open operation which are common to all file organisations, such as volume-id checking, are described in detail in section 9.

9.3.1 File Conditions

The file not found ($$RES = 3) or wrong type ($$RES = "1") condition is signalled if a file with Speedbase organisation and the same file-id as that specified in the FD is not present on the direct access volume. The index not present ($$RES = "2") condition is signalled if the index whose name is specified is not defined in the database file.

9.3.2 Successful Completion
Providing no file condition or irrecoverable I/O error occurs, the OPEN operation completes successfully. The index specified in SPINDEX is found, and access to the record set to which it applies is made possible via the appropriate key. The file size and record length are returned in the FD and made available to the user program if the SIZE and RECORD LENGTH statements with the symbol option were coded.

9.3.3 Programming Notes
The database index file and all the data record files will be opened by the OPEN operation. This means that the open operation will in fact open up to four channels. In addition the open operation will open and close the database dictionary to obtain the index information.

An I/O error or not found error can refer to any one of these files.

9.4 The READ statement
READ is used to retrieve a record with a given key from the file. If the key is not present the record area will not be changed.

The READ statement is coded:

```
READ filename  INTO A [lock-type]
```

Here filename identifies the Speedbase database file definition, A is a simple or indexed variable and lock-type is one of LOCK or PROTECT as specified earlier. If READ is attempted on an FD which is not already open the program will be terminated in error.

9.4.1 Establishing the Key
The key is established by setting the various elements of the key within the record area before the READ statement is executed.

9.4.2 File Conditions
The record not found file condition will be signalled if a record with the key you have specified is not present on the file. In this case the record area will remain unchanged. The "virtual position" in the file, for subsequent READ NEXT and READ PRIOR operations, will be set as if the record specified by the key value of the failing READ does exist.

If the lock-type clause is coded Global System Manager will attempt to obtain the appropriate lock for you. If this is not possible, for example due to some other user already having a competing lock himself, then the lock unavailable condition ($$COND = 3) is signalled. The record will still be retrieved, but will have not been locked.

9.4.3 Successful completion
Providing no permanent I/O error occurs and a key equal to the one you specified exists on the file, READ will transfer bytes from the record thus identified to A. The number of bytes transferred does not depend on the picture clause associated with A, but will be equal to the record length of the record set defined when the file was created.

9.5 The READ NEXT and READ PRIOR statements

READ NEXT and READ PRIOR are used to process part or all of a file sequentially. READ NEXT processes keys in ascending sequence, and READ PRIOR processes keys in descending sequence. They are coded:

```
     READ NEXT filename [KEY LENGTH length] INTO A [lock-type]
and:
     READ PRIOR filename [KEY LENGTH length] INTO A [lock-type]
```

Here filename identifies the Speedbase file definition, length is an integer literal in the range 1 to 50, A is a simple or index variable and lock-type is one of LOCK or PROTECT. If a READ NEXT or READ PRIOR is attempted on an FD which is not already open the program will be terminated in error.

9.5.1 The Record Retrieved

The record retrieved by READ NEXT or READ PRIOR depends on the previous file operation:

- A READ NEXT following an OPEN will retrieve the first record of the record set. If there are no records in the record set end of file will be signalled.

- A READ NEXT following a successful READ, READ NEXT, READ PRIOR, READ FIRST or READ LAST will retrieve the record immediately higher in collating sequence than the one just read processed. If there is no such record end of file will be signalled.

- A READ NEXT following an unsuccessful READ, READ FIRST or READ LAST will retrieve the record immediately higher in collating sequence than the virtual, non-existent record that was specified by the previous READ. If there is no such record end of file will be signalled.

- A READ PRIOR following a successful READ, READ NEXT, READ PRIOR, READ FIRST or READ LAST will retrieve the record immediately lower in collating sequence than the one just processed. If there is no such record start of file will be signalled.

- A READ PRIOR following an unsuccessful READ, READ FIRST or READ LAST will retrieve the record immediately lower in collating sequence than the virtual, non-existent record that was specified by the previous READ. If there is no such record start of file will be signalled.

- A READ PRIOR following an OPEN will signal start of file immediately.

- The record retrieved by a READ PRIOR or READ NEXT operation following a READ PHYSICAL operation will be dependent on the operation prior to the READ PHYSICAL operation as above. The read physical operation does not affect the sequencing.

Where a KEY LENGTH clause was specified, the partial key established serves to delimit the range of the READ NEXT or READ PRIOR operation. If the key of the record retrieved does not match the partial key established in the record area the key match condition will be signalled.

9.5.2 File conditions

The end of file condition ($$COND = 2) will be signalled in response to READ NEXT if the record last accessed was the record with the highest key, or if the last file operation encountered end of file. When this condition occurs the record area is unchanged. The start of file condition ($$COND = 2) will be signalled in response to READ PRIOR if the last record accessed was the record with the lowest key, or if the last file operation encountered start of file. Additionally the key check condition ($$COND = 2) will be signalled if the partial key established in the record area prior to the READ NEXT or READ PRIOR did not match the start of the key in the record retrieved. In this latter case record area will remain unchanged.

If the lock-type clause was coded Global System Manager will attempt to obtain the appropriate lock for you. If this is not possible due to some other user already having a competing lock, then the lock unavailable ($$COND = 3) is signalled. The record will still be retrieved, but will not have been locked.

Note that if the key match condition has been signalled then the record will not be locked, regardless of any lock option which might be specified.

9.5.3 Successful Completion

Providing no irrecoverable I/O error occurs, bytes will be transferred from the file to A. The number of bytes transferred will not depend on the picture clause of A, but will be equal to the record length of the file, defined when it was created.

9.5.4 Programming Notes

Unlike the FETCH NEXT statement in Speedbase the record area is not set to HIGH-VALUES when an end of file condition is reached but is set to the highest key value.

9.6 The READ FIRST and READ LAST statements

READ FIRST and READ LAST are used to retrieve the first or last record of a record set. They are coded:

```
     READ FIRST filename [KEY LENGTH length] INTO A [lock-type]
and:
     READ LAST filename [KEY LENGTH length] INTO A [lock-type]
```

Here filename identifies the Speedbase file definition, length is a numeric literal in the range 1 to 50, A a simple or indexed variable and lock-type is one of LOCK or PROTECT. If a READ FIRST or READ LAST is attempted on an FD which is not already open the program will be terminated in error.

9.6.1 The Record Retrieved

The record retrieved by READ FIRST or READ LAST depends on the partial key specified by the KEY LENGTH clause. If no partial key is specified either the very first or the very last record of the record set is retrieved. If a partial key is specified the first or last record in the record set whose key start matches the partial key as specified in the record area is retrieved.

9.6.2 File Conditions

The end of file condition ($$COND = 2) will be signalled in response to READ FIRST if there are no records in the record set. The start of file condition ($$COND = 2) will be signalled in response to READ LAST if there are no records in the record set. In both cases the record area will remain unchanged. The "virtual position" in the file, for subsequent READ NEXT and READ PRIOR operations, will be set as if the record specified by the key value of the failing READ does exist.

Additionally the key check condition ($$COND = 2) will be signalled if there is no record in the record set whose key start matches the partial key specified in the record area. In this case the record area will also remain unaffected by the operation.

If the lock-type clause was coded System Manager will attempt to obtain the appropriate lock for you. if this is not possible, for example due to some other user already having a competing lock himself, then the lock unavailable condition ($$COND = 3) is signalled. The record will still be retrieved, but will not have been locked.

9.6.3 Successful Completion

Providing no error condition occurs, bytes will be transferred from the file to A. The number of bytes transferred will not depend on the picture clause of A, but will be equal to the record length of the record set defined when the file was created.

9.7 The READ PHYSICAL statement

READ PHYSICAL is used to retrieve a record with a given relative record number. You code:

```
READ PHYSICAL filename INTO A [lock-type]
```

Here filename identifies the Speedbase file definition, A is a simple or indexed variable and lock-type is one of LOCK, or PROTECT. The relative record number must have been established in the SPRNN field prior to the READ PHYSICAL operation being issued. If READ PHYSICAL is attempted on an FD which is not already open the program will be terminated in error.

9.7.1 File Conditions

If the READ PHYSICAL attempts to return a deleted record then the record deleted condition ($$COND = 4) is signalled.

If the lock-type clause was coded System Manager will obtain the appropriate lock for you. If this is not possible, for example due to some other user already having a competing lock, then the lock unavailable condition ($$COND = 3) is signalled. The record will still be retrieved, but will not be locked.

9.7.2 Successful completion

Providing no error occurs, bytes will be transferred from the file to A. The number of bytes transferred will not depend on the picture clause associated with A but will be equal to the record length of the record set, defined when it was created.

9.7.3 Programming Notes

When a READ PHYSICAL operation is performed, the data is transferred from the record with the relative record number established by the

calling program. No checking is performed to establish the fact that this is a valid relative record number. It is being assumed that the calling program has saved the relative record number correctly. It is also the responsibility of the calling program to perform suitable validation checks on the record retrieved before processing it, as it may have been deleted, reused or updated since its number was saved.

Normally after a file processing statement has completed the relative record number field will contain the record number of the record last processed. However after a start of file or end of file condition has been signalled (and immediately after an OPEN operation) the SPRNN value will not be valid. Programs should not attempt a READ PHYSICAL with these values.

It is generally inappropriate to keep a relative record number from one run of a program to another, as a reorganisation of the database which might be performed between the two program runs.

An example of a situation in which a READ PHYSICAL might be useful is in a sort. Records would be read from the database using READ NEXT, and the sort key constructed from them, with the relative record number appended as in a tag sort. When the sorted records are returned READ PHYSICAL is used to retrieve the records, thereby avoiding any index processing and speeding the whole process considerably.

It should be noted that a record may have been updated between these two accesses, or deleted, or even reused by an application using Speedbase Presentation Manager. It is therefore essential that either the entire process is carried out so that no such interference can occur (by having the file OPENed OLD for example), or that the returned record is checked to ensure its validity. In practice if a record has been updated so as to no longer be valid, you will probably have to ignore it.

9.8 The UNLOCK statement

The UNLOCK statement is used to release any locks outstanding from a previous READ type operation using the lock-type clause. You code:

```
UNLOCK filename
```

where filename identifies the Speedbase file definition. If an UNLOCK is attempted on an FD which is not already open the program will be terminated in error.

9.8.1 Successful completion

Any outstanding lock on the file due to a previous READ type operation on the FD passed to the UNLOCK is released.

Note particularly that any lock gained through the use of the LOCK verb is not released by this UNLOCK statement, but must be released by using an UNLOCK filename region statement as documented in this manual.

9.9 The CLOSE Statement

CLOSE must be used to terminate the processing of a file. It is coded:

```
CLOSE filename
```

where filename identifies the Speedbase file definition. If a CLOSE is attempted on an FD which is not already open the program will be terminated in error.

9.9.1 Standard Processing

CLOSE always completes any outstanding I/O operations on the file and returns the FD to the status it possessed prior to being opened. Following CLOSE, the FD can be re-opened for the same or a different Speedbase file using the same or a different index, by a subsequent OPEN statement.

9.9.2 Programming Notes

CLOSE on a Speedbase file releases any locks outstanding on the particular FD closed, which were gained through use of the Speedbase lock option on a READ type statement. CLOSE does not release any other locks (including locks on other FD's, or locks gained through the use of the LOCK verb), unless the CLOSE releases the last active channel on the database file. In practice this means you should exercise extreme caution in using the LOCK verb with a Speedbase database, and should ensure that, and locks you obtain using it, are explicitly released by using the UNLOCK verb.

The CLOSE verb closes the index files and all the data files of the database. An I/O error returned can be due to closing any one of these files.

9.10 Speedbase compatibility

The operation code for the read only access method is compatible with the operational code for Speedbase Presentation Manager. Databases accessed by the Speedbase access method can be accessed concurrently by Speedbase Presentation Manager.

The locking is compatible. The PROTECT lock types are equivalent, and the LOCK lock-type being the same as the default locking in Presentation Manager.

9.11 Memory Paged SPAM

This version of SPAM has most of the SPAM code within a System Manager memory page and therefore requires less space within a program. It cannot be used under pre-V8.1 Global System Manager.

Paged SPAM operates in exactly the same way as non-paged SPAM. No changes are required to any code using standard SPAM, all that is needed is that C.$PAGES is linked in to your program before C.$APF and C.$MCOB ensuring that module AS$Z is linked in preference to module AS$R. (See section on memory paged subroutines in the System Subroutines manual).

9.12 The Speedbase Access Method for Speedbase in C-ISAM

This version of the read only Speeedbase access method, can only be used on Speedbase databases where the data is held in C-ISAM databases. Please refer to your Speedbase Development Manual for details on the file structure.

9.12.1 Record Format

The record format for SPAM for C-ISAM is described in section 9.1.1 and also in the Speedbase Development Manual.

9.12.2 ORGANISATION statement

The organisation statement for SPAM in C-ISAM is as follows:

```
ORGANISATION OR$99C TYPE 99 EXTENSION 144
```

9.12.3 The File Definition

The file definition is as described in section 9.2 except that the FD statement must be as follows:

```
FD filename ORGANISATION OR$99C
```

9.12.4 The File Processing Statements

The file processing statements are as described in sections 9.3 to 9.9.

If an I/O error occurs on the Speedbase C-ISAM database the Unix error code will be returned in the system variable $$CRES, a PIC 9(6) COMP field. For information about the Unix error code see your C-ISAM manual from INFORMIX or Unix Manuals from your operating system supplier.

9.12.5 Programming Notes

All locking on the Speedbase file will be done on the Speedbase schema file within System Manager. This does mean that the Speedbase C-ISAM database must not be accessed by other Unix applications at the same time as it is being accessed by Global applications.

For further information about accessing Speedbase C-ISAM files please refer to the Speedbase Development Manual.

9.13 The Speedbase Access Method for Speedbase in Btrieve

This version of the read only Speeedbase access method, can only be used on Speedbase databases where the data is held in Btrieve databases. Please refer to your Speedbase Development Manual for details on the file structure.

9.13.1 Record Format

The record format for SPAM for Btrieve is described in section 9.1.1 and also in the Speedbase Development Manual.

9.13.2 ORGANISATION statement

The organisation statement for SPAM in Btrieve is as follows:

```
ORGANISATION OR$99N TYPE 99 EXTENSION 144
```

9.13.3 The File Definition

The file definition is as described in section 9.2 except that the FD statement must be as follows:

```
FD filename ORGANISATION OR$99N
```

9.13.4 The File Processing Statements

The file processing statements are as described in sections 9.3 to 9.9.

If an I/O error occurs on the Speedbase Btrieve database the Unix error code will be returned in the system variable $$CRES, a PIC 9(6) COMP field. For information about the Btrieve error code see your Btrieve manual from your system supplier.

9.13.5 Programming Notes

There is no Global locking required for Btrieve SPAM , and locking is not available.

For further information about accessing Speedbase C-ISAM files please refer to the Speedbase Development Manual.

9.14 Memory Paged Open SPAM

This version of SPAM is memory paged and can be used on the Global, Btrieve and C-ISAM versions of Speedbase. It cannot be used under pre-V8.1 (revision f) Global System Manager.

The organisation statement is as for non-paged SPAM for Global Speedbase files. It behaves in the same way as the appropriate non-paged routine for the individual types of Speedbase databases.

In order to use open SPAM you must insure that your program is linked in the following way:

```
$44 LINK:database program      UNIT:SSS
$44 LINK:C.$PAGES/AW$Z UNIT:$S
$44 LINK:C.$PAGES UNIT:$S
```

ensuring that module AW$Z is linked in preference to modules AS$R and AS$Z. (See section on memory paged subroutines in the System Subroutines manual).

10. The C-ISAM Indexed Sequential File Organisation

10.1 C-ISAM Indexed Sequential Files

The C-ISAM indexed sequential access method (CIAM) can be used to read and write to a C-ISAM database, on a single index, from the Global System Manager environment. To do this it is necessary to create a CIAM schema file in a Global directory using RCBUILD and $SETIRU (see chapters 13 and 14). The schema file contains the details of the C-ISAM database you want to access and all file operation are done via the schema file. (See your C-ISAM Programmer's guide for details of C-ISAM files.)

10.1.1 Specifying File Attributes

The attributes of the schema file, such as its unit-id, volume-id and file-id , are specified in its file definition (FD), coded in the data division.

The index will be as defined by the RCBUILD record conversion table as defined when creating the schema file.

10.1.2 File Processing Statements

The following procedure division statements are provided to enable a C-ISAM database, through the schema file, to be processed. They are:

OPEN	must be executed prior to any either statement affecting the file;
READ	retrieves a specific record corresponding to a specified index key value;
READ NEXT	retrieves the next record in the sequence of the specified index;
READ PRIOR	retrieves the preceding record in the sequence of the specified index;
READ FIRST	retrieves the first record in the sequence of the specified index;
READ LAST	retrieves the last record in the sequence of the specified index;
READ PHYSICAL	retrieves a record using its record number;
WRITE	writes a new record to the database;
REWRITE	updates the record just read;
UNLOCK	is used to release a lock obtained by a previous READ operation;
CLOSE	must be issued to terminate file processing.

When any of the statements is executed a file condition, signalled by an exception condition 2, may arise as explained in 1.2.1. In addition, if OPTION ERROR is specified in the FD, exception condition

1 will be generated should an irrecoverable I/O error occur. In either case any C-ISAM or Unix error will be returned in the System Variable $$CRES, a PIC 9(6) COMP field. It is therefore usual to follow each file processing statement with an ON EXCEPTION statement. If you do not , and an exception condition arises, your program will terminate in error.

10.1.3 Data Records

The data record definition used by the application must be in Global format as defined by the RCBUILD conversion table used in creating the schema file and not in the C-ISAM format.

10.1.4 File locking

The C-ISAM access method provides a lock option which is specified by appending the lock-type, LOCK, to any of the READ, READ NEXT, READ PRIOR, READ FIRST, READ LAST, READ PHYSICAL statements. The lock verb will attempt to get an exclusive lock on the schema file for the system (e.g. system 1B) being used. If another partition on the system has the file locked, then a lock exception will be returned. The LOCK operation will then attempt to obtain a C-ISAM lock on the record returned.

Any number of locks can then be issued by the partition on the system with locking control.

An UNLOCK operation will release all the C-ISAM locks for that partition as well as the lock on the schema file for the FD. If several FD's are being used in a single program at the same time to access the same schema file, then although an UNLOCK operation on one FD will release all the C-ISAM locks, it will not release all the Global locks on the schema file for every FD in the partition. UNLOCK must therefore be called for every FD.

If more than one schema file is being used to access a single C-ISAM database, then care must be taken to ensure that locks on both files are issued, otherwise the C-ISAM file will not be locked against all Global access.

10.1.5 The ORGANISATION Statement

The following statement must be coded in the data division before the first FD declaration:

```
ORGANISATION OR$96 TYPE 1 EXTENSION 16
```

10.2 The File Definition

The file definition for the schema file is coded in either working storage or in the linkage section as follows:

```
FD filename ORGANISATION OR$96
[ASSIGN TO UNIT unit-id FILE file-id [VOLUME volume-id]
[KEY IS keyname]
[OPTION ERROR]
[ON ERROR intercept]
```

The FD establishes a special group data item, 96 bytes in length whose name is filename. The quantities unit-id, file-id, volume-id and intercept appear as subordinate items within this and can, if need be, be referred to by the application program.

If it is possible to specify unit-id, file-id or volume-id before the program executes then the quantity should be coded as a character string in quotes. If any of these quantities are not known until run-time then a symbol must be quoted for the quantity. The symbol will then label a 02 level item which the user program is responsible for initialising.

10.2.1 The Filename
The filename must be a symbol. It serves to label the file definition, as explained in section 1 of the Global Cobol File Management Manual.

10.2.2 The ORGANISATION Clause
The organisation clause must be coded as shown, except that you may use the abbreviation ORG instead of ORGANISATION. It indicates that the file is a CIAM schema file.

10.2.3 The ASSIGN Statement
Use of the ASSIGN statement, which is the same for any file organisation is explained in section 1.

10.2.4 Optional Statement Placements
The KEY IS, ON ERROR and OPTION ERROR statements are optional and may appear in any order following the ASSIGN statement.

10.2.5 The KEY Statement
The KEY statement is only required if records are to be accessed using the READ PHYSICAL OPERATION. Keyname is specified as a symbol and the statement:

```
02 symbol PIC 9(9) COMP
```

is generated within the FD.

CIAM maintains the keyname field to contain the record number of the C-ISAM record last accessed. The program must place the record number in the keyname field before executing a READ PHYSICAL operation.

A successful READ, READ NEXT, READ PRIOR, READ LAST, READ FIRST or WRITE operation will return the C-ISAM record number of the record accessed in the keyname field.

10.2.6 The OPTION ERROR and ON ERROR statements
OPTION ERROR should be coded only if you wish your program to regain control following an irrecoverable I/O error. ON ERROR should be coded if you wish to handle certain I/O errors specially. The processing of these statements is common to all file organisations, and is described in section 1.

10.3 The OPEN Statement
The OPEN statement is coded:

```
OPEN type filename
```

where type id the word OLD or SHARED and filename identifies the CIAM file definition.

An OPEN statement must be executed before any other file processing statement. If an OPEN is attempted but the FD is already open your program will terminate in error.

OPEN NEW is not supported because the C-ISAM file must always be created by another C-ISAM application and the Global schema file by $SETIRU. OPEN OLD obtains exclusive access to the schema file and opens the C-ISAM file whilst the OPEN SHARED allows co-operating jobs to share a schema file. (The features of the open operation which are common to all file organisations such as volume-i checking are described in detail in section 1.2 of this manual.)

10.3.1 File Conditions

The file not found ($$RES = "3") or wrong type ($$RES = "1") condition is signalled if the schema file is not present or of the wrong file organisation or if the C-ISAM file is not present on the directory specified by the schema file. The index not present ($$RES = "2") condition is signalled if the index defined in the schema file is not present on the C-ISAM file.

10.3.2 Successful Completion

Providing no file condition or irrecoverable I/O error occurs, the OPEN operation completes successfully. The index specified in the schema file is found and access to the C-ISAM file is made possible via the appropriate key.

10.4 The WRITE Statement

WRITE is used to create a new record identified by the supplied record area to the C-ISAM file. It is coded:

```
WRITE filename FROM A
```

Here filename identifies the CIAM file definition, and A may be a simple or indexed variable. If WRITE is attempted on an FD which is not open then the program will terminate in error.

10.4.1 File Conditions

CIAM attempt to write a new record to the C-ISAM database. If any unique index contains a key of the same value then a key already exists condition ($$RES = 4) will be returned.

10.4.2 Successful Completion

Providing that no file condition or irrecoverable I/O error occurs, WRITE will either cause an overflow record to be created or an existing record to be modified. The number of bytes transferred from the record at A to direct access storage will not depend on the picture clause of A but will be equal to the record length specified in the schema file.

10.5 The REWRITE Statement

REWRITE is used to update the record last accessed from a C-ISAM database. It is coded:

```
REWRITE filename FROM A
```

Here filename identifies the CIAM schema file definition and A is a simple or indexed variable or literal. If REWRITE is attempted on an FD which is not already open the program will terminate in error.

10.5.1 File Conditions

If the key value of any unique index is modified the unique key condition ($$COND = 4) will be returned. In this case, if you want to update the record, the record must be deleted using the DELETE operation and then written using the WRITE statement.

10.5.2 Successful Completion

Providing the key has not been erroneously modified, and no irrecoverable error occurs, REWRITE will update the record last accessed. The number of bytes converted and transferred from the record at A to direct access storage will not depend on the picture clause of A but will be equal to the record length specified in the schema file.

10.6 The DELETE statement

The DELETE statement is used to delete the last record processed. You code:

```
DELETE filename
```

where filename identifies the CIAM schema file definition. If the FD is not already open then your program will terminate in error.

10.6.1 File Conditions

No file condition can arise as a result of the DELETE statement.

10.6.2 Successful Completion

Providing that no irrecoverable error occurs the record will be deleted from the database file.

10.7 The READ statement

READ is used to retrieve a record with a given key from the C-ISAM database. If the key is not present the record area will not be changed.

The READ statement is coded:

```
READ filename INTO A [LOCK]
```

Here filename identifies the schema file for the C-ISAM database and A is a simple or indexed variable. If READ is attempted on an FD which is not already open the program will terminate in error.

10.7.1 Establishing the Key

The key is established by setting the various elements of the key within the record are in Global format before the READ statement is executed.

10.7.2 File Conditions

The record not found condition will be signalled if a record with the key you have specified is not present on the file. In this case the record area will remain unchanged. If the LOCK clause was coded an

attempt will be made to obtain a lock on the schema file and to obtain a C-ISAM lock o the record. If this is not possible a lock unavailable error ($$COND = 3) will be returned and the record will not be retrieved.

10.7.3 Successful Completion

Providing no permanent I/O error occurs and the record can be locked if the LOCK clause is specified, READ will convert and transfer bytes from the record thus identified to A. The number of bytes transferred does not depend on the picture clause of A, but will be equal to the record length as defined in the schema file.

10.8 The READ NEXT and READ PRIOR Operations

READ NEXT and READ PRIOR are used to process part or all of a file sequentially. READ NEXT processes keys in ascending sequence, and READ PRIOR processes keys in descending sequence. They are coded:

```
     READ NEXT filename INTO A [LOCK]
and:
     READ PRIOR filename INTO A [LOCK]
```

Here filename identifies the CIAM schema file definition and A is a simple or indexed variable. If a READ NEXT or READ PRIOR is attempted on an FD which is not already open the program will terminate in error.

10.8.1 The Record Retrieved

The record retrieved by READ NEXT or READ PRIOR depends on the previous file operation:

- A READ NEXT following an OPEN statement will retrieve the first record of the C-ISAM file. If there are no records in the record set end of file will be signalled.

- A READ NEXT following a READ, READ NEXT, READ FIRST, READ LAST, READ PHYSICAL, WRITE or DELETE will retrieve the record immediately higher in collating sequence than the one just read processed. If there is no such record end of file will be signalled.

- A READ PRIOR following a READ, READ NEXT, READ FIRST, READ LAST, READ PHYSICAL, WRITE, REWRITE or DELETE will retrieve the record immediately lower in collating sequence then the one just processed. If there is no such records start of file will be signalled.

- A READ PRIOR following an OPEN will signal start of file immediately.

10.8.2 File Conditions

The end of file condition ($$COND = 2) will be signalled in response to READ NEXT if the record last accessed was the record with the highest key, or if the last file operation encountered end of file. When this condition occurs the record area remains unchanged. The start of file condition ($$COND = 2) will be signalled in response to a READ PRIOR if the last record accessed was the record with the lowest key, or if the last file operation encountered start of file. In this case the record area will remain unchanged.

If the LOCK clause was coded an attempt will be made to obtain a lock on the schema file and to lock the C-ISAM record. If this is not possible then the lock unavailable condition ($$COND = 3) is signalled and the record will not be retrieved

10.8.3 Successful Completion

Providing no irrecoverable error occurs, bytes will be converted and transferred from the file to A. The number of bytes transferred will not depend on the picture clause associated with a, but will be equal to the record length defined in the schema file.

10.9 The READ FIRST and READ LAST statements

READ FIRST and READ LAST are used to retrieve the first or last record of a record set. They are coded:

```
     READ FIRST filename INTO A [LOCK]
```

and:

```
     READ LAST FILENAME INTO A [LOCK]
```

Here filename identifies the CIAM schema file definition and A is a simple or indexed variable. If a READ FIRST or READ LAST is attempted on an FD which is not already open the program will terminate in error.

10.9.1 The Record Retrieved

The record retrieved by READ FIRST or READ LAST will be the very first or the very last record of the index specified in the schema file on the C-ISAM database.

10.9.2 File Conditions

The end of file condition ($$COND = 2) will be signalled in response to READ FIRST if there are no records in the record set. The start of file condition ($$COND = 2) will be signalled in response to a READ LAST if there are no records in the record set.

If the LOCK clause was coded and the attempt to obtain as lock failed then the lock unavailable condition ($$COND = 3) is signalled. The record will not be retrieved.

10.9.3 Successful Completion

Providing no irrecoverable I/O error occurs bytes will be converted and transferred from the file to A. The number of bytes transferred will not depend on the picture clause associated with A, but will be equal to the record length specified in the schema file.

10.10 The READ PHYSICAL statement

READ PHYSICAL is used to retrieve a record with a given record number provided that the C-ISAM file has a record numbering index defined. You code:

```
     READ PHYSICAL filename INTO A [LOCK]
```

Here filename identifies the CIAM schema file definition and A is a simple or indexed variable. The record number must have been established in the KEY IS clause prior to the READ PHYSICAL operation

being issued. If the READ PHYSICAL is attempted on an FD which is not already open the program will be terminated in error.

10.10.1 File Conditions

If the READ PHYSICAL is attempted on an invalid record number for which there is no record, the record not found condition ($$COND = 2) will be signalled.

If the LOCK clause was coded an the attempt to gain the lock failed, then the lock unavailable condition ($$COND = 3) is signalled. The record will not be retrieved.

10.10.2 Successful Completion

Providing no irrecoverable I/O error occurs and any lock does not fail, bytes will be converted and transferred from the file to A. The number of bytes transferred will not depend on the picture clause of A but on the record length specified in the schema file.

10.11 The UNLOCK statement

The UNLOCK statement is used to release any locks outstanding on the C-ISAM file for the current system and to release the lock on the current FD for the schema file. You code:

```
UNLOCK filename
```

where filename identifies the CIAM schema file definition. If the UNLOCK is attempted on an FD which is not already open the program will be terminated in error.

10.11.1 Successful Completion

All locks on the C-ISAM file for this system will be released and the lock on the schema file for this FD will also be release. Note that an UNLOCK does not release all schema file locks for this system and that this will need to be done explicitly for each FD.

Note particularly that the lock gained through use of the LOCK statement is not released by this UNLOCK statement, but must be release using an UNLOCK filename region statement.

10.12 The CLOSE Statement

CLOSE must be used to terminate the processing of a file. It is coded:

```
CLOSE filename [DELETE]
```

where filename identifies the CIAM schema file definition. If a CLOSE is attempted on an FD which is not already open the program will be terminated in error.

10.12.1 Standard Processing

CLOSE always completes any outstanding I/O operations on the schema file and C-ISAM file and returns the FD to the status it possessed prior to being opened. Following a CLOSE, the FD can be re-opened by a subsequent OPEN statement.

10.12.2 Deletion

If the DELETE phrase is coded all the space the schema file and C-ISAM database occupies is returned to the volume and the file-id and name is erased from the directory.

WARNING: A CLOSE DELETE will delete the entire database file. This option should be used with extreme caution.

11. The Direct Unix File Organisation

The direct Unix file organisation allows access to Unix files from the Global System Manager environment and may therefore only be run on Global System Manager Unix. The Unix file accessed is treated as a string of bytes numbered sequentially from zero upwards. Records are accessed by supplying the starting byte number as key, together with the number of bytes to be read or written.

11.1 File Structure

11.1.1 Record Format

The format of the Unix file records is entirely under program control since you determine the length and starting byte of each record.

11.1.2 Specifying File Attributes

The attributes of a file, such as its unit-id, volume-id and file-id, specified in the file definition (FD) coded in the data division has no meaning but must be specified. It is recommended that the file-id is set to a descriptive variable name, the unit-id to "?" and the volume-id should not be set at all., coded in the data division.

You must specify the name of the area which contains the record length. You can also define a key area to contain the starting byte number for use in random access READ and WRITE statements. Section 11.2 below describes those parts of the file definition which are specific to the basic direct file organisation, but the statements which are common to all organisations are defined in section 1.2.

11.1.3 File Processing Statements

A number of procedure division statements are provided to enable a Unix file to be processed. They are:

OPEN which must be executed prior to any other statement affecting the file;

READ to read a record at random;

READ FIRST to read the very first record in the file;

READ LAST to read the very last record in the file;

READ NEXT to read the next record during sequential processing;

READ PRIOR to read the previous record during sequential processing;

WRITE to update an existing record at random;

WRITE NEXT to write a record during sequential processing;

CLOSE to terminate processing of a file.

When any of the statements is executed a file condition, signalled by exception condition 2, may arise as explained in section 1.3.1. In addition, if OPTION ERROR or ON ERROR is specified in the FD, an

exception condition will be generated should an irrecoverable I/O error occur, as explained in section 1.6. Therefore it is usual to follow each file processing statement with an ON EXCEPTION statement. If you do not, and an exception condition arises, your program will be terminated in error.

11.1.4 Extent Boundary Checking

A write next pointer is not maintained for a direct Unix file, and you may use WRITE NEXT, WRITE, READ FIRST, READ LAST, READ NEXT, READ PRIOR and READ statements to access information anywhere within the allocated file extent. However, should any of these statements attempt to read or write a record which is partially or wholly outside the file extent, the file boundary violation file condition will be signalled.

11.1.5 The ORGANISATION Statement

If a program uses the basic direct access method then the statement:

```
ORGANISATION OR$97 TYPE 0 EXTENSION 128
```

must be coded in the data division before the first FD or data declaration.

11.2 The File Definition

The file definition for a Unix direct file is coded in either working storage or the linkage section as follows:

```
FD filename ORGANISATION OR$97
[ASSIGN TO UNIT "?" FILE file-id ]
[KEY IS keyname]
RECORD LENGTH IS length
[SIZE IS size]
[OPTION ERROR]
[ON ERROR intercept]
01 FILLER REDEFINES filename
02 FILLER PIC X(88)
02 PANAME PIC X(99) * Unix directory path and filename
02 FILLER PIC X
VALUE #00
```

The FD establishes a special group data item, 208 bytes in length, whose name is filename. The quantities file-id, Unix

Directory path and filename, keyname, length and size appear as subordinate items within this group and can, if need be, be referred to by the application program.

It is possible to specify the Unix path and file name (which must be terminated with a #00 byte) before the program executes, In this case the quantity should be coded with a VALUE clause as a character string. If you can specify the size or length before the program executes, code the quantity as a numeric string. If the Unix directory path and filename is not known until run-time then the quantity must be moved to the PANAME field during program execution. If the size or length is not known until run-time a symbol must be coded for the quantity. This symbol will then label a level 02 item which the user program is responsible for initialising.

11.2.1 The Filename
The filename is coded as a descriptive symbol. It serves to label the file definition, as explained in section 1.2.1.

11.2.2 The ORGANISATION Clause
The ORGANISATION clause must be coded as shown. It indicates that the basic direct access method is to be used.

11.2.3 The ASSIGN Statement
The ASSIGN statement is required in working storage and should be coded as shown, where file-id is only required as a descriptive symbol.

11.2.4 Optional Statement Placement
The KEY, RECORD LENGTH, SIZE, OPTION and ON ERROR statements may appear in any order following the ASSIGN statement. The RECORD LENGTH statement must be coded: the others are optional.

11.2.5 The KEY Statement
The KEY statement is only required if records are to be accessed at random. Keyname is specified as a symbol and the statement:

```
02 symbol PIC 9(9) COMP
```

is generated within the FD.

DUAM maintains the keyname field to contain the byte number of the start of the record last accessed. The byte number of the very first byte of the file counts as byte number zero.

The program must place the byte number of the start of the record it requires to access in the keyname field before executing a random READ or WRITE operation.

A successful READ NEXT or WRITE NEXT operation increments the keyname field by the length of the previous record accessed and then accesses the record thus identified: if the previous operation on the file was an OPEN then the first record is accessed.

A successful READ PRIOR decrements the keyname field by the length of the record to be read, and then accesses that record.

11.2.6 The RECORD LENGTH Statement
The RECORD LENGTH statement is always required. Length is specified as a symbol and the statement:

```
02 symbol PIC 9(4) COMP
```

is generated within the FD.

The program must place the length of the record to be read in the length field before executing a read or write statement. The field is set to zero when the file is opened, and is not altered by read and write operations.

11.2.7 The SIZE Statement
The SIZE statement is required only if you wish know the current size of the file. The file size will be maintained during processing.

11.2.8 The OPTION and ON ERROR Statements

OPTION ERROR should be coded only if you wish your program to regain control following an irrecoverable I/O error. ON ERROR should be coded if you wish to handle certain I/O errors specially. The processing of these statements is common to all file organisations, and is described in section 1.6.

11.3 The OPEN Statement

The OPEN statement is coded:

```
OPEN type filename
```

where type is the word NEW or OLD filename identifies the direct Unix file definition. An OPEN statement must be executed before any other operation affecting the file. If an OPEN is attempted but the FD is already open your program will be terminated in error.

OPEN NEW is used to create a new Unix file of the specified path and file name if one does not already exist or to truncate a file that already exists to zero length. If there is no existing file the new file will be create with Unix permissions 600 octal (i.e. RW access for the owner). OPEN OLD obtains exclusive access to an existing file.

11.3.1 File Conditions

When an OPEN OLD statement is executed the System Manager checks to see whether a file with the same Unix path and file name exists. If this is not the case, file not found is signalled.

11.3.2 Successful Completion

Providing no file condition or irrecoverable I/O error occurs an OPEN NEW results in an empty file of size zero being opened on with the specified Unix path and file name.

When an OPEN OLD statement completes successfully the file size is returned in the FD and can be accessed by the user program if a SIZE statement with the symbol option was coded.

The record length field in the FD is set to zero by a successful OPEN statement.

The keyname field in the FD is always set to zero following a successful OPEN statement. This is to allow the file to be processed sequentially using a sequence of READ NEXT or WRITE NEXT statements as described below.

11.4 The WRITE NEXT Statement

WRITE NEXT is used to write all or part of a file sequentially. It is coded:

```
WRITE NEXT filename FROM A
```

Here filename identifies the Direct Unix file definition and A is a simple or indexed variable. If a WRITE NEXT is attempted on an FD which is not already open the program will be terminated in error.

11.4.1 Establishing the Length

The FD must contain the RECORD LENGTH statement with the length coded as a symbol. The program must establish the length of the record to be written (1 to 32767 bytes) in this field before executing the WRITE NEXT.

If the length is not established explicitly by the program the length of the last record accessed will be used.

11.4.3 Successful Completion

Provided that no file condition or irrecoverable I/O error occurs WRITE NEXT will set the key to the number of the byte following the previous record accessed (if any) and transfer A to the record thus identified. The number of bytes transferred is given by the length field.

11.4.4 Programming Note

If a KEY statement was not specified in the file definition the FD still contains an internal key field which is set to zero when the file is opened. In this case WRITE NEXT can be used to write the entire file sequentially.

When a KEY statement is specified a READ or WRITE can be used to position the file at any point. WRITE NEXT will continue writing from the record following the one accessed by READ or WRITE.

The READ NEXT statement sets the key field to the record retrieved, so that a subsequent WRITE NEXT statement will write the following record.

The WRITE NEXT can cause the file size to be extended if needed, subject to space being available in the Unix file system.

11.5 The WRITE Statement

WRITE is used to write a record at random or rewrite the last record accessed. It is coded:

```
WRITE filename FROM A
```

Here filename identifies the direct Unix file definition and A is a simple or indexed variable. If a WRITE is attempted on an FD which is not already open the program will be terminated in error.

11.5.1 Establishing the Key

If WRITE is to be used as a random access operation the FD must contain a KEY statement coded as:

```
KEY IS symbol
```

which causes:

```
02 symbol PIC 9(9) COMP
```

to be generated. The program must then move the byte number of the start of the record it is required to retrieve to this field before executing the WRITE statement.

If a KEY statement is not coded in the file definition the FD still contains an internal key field which is set to zero when the file is

opened and incremented by READ NEXT and WRITE NEXT. In this case the only use of the WRITE statement is to rewrite the record previously retrieved by READ NEXT or written by WRITE NEXT.

11.5.2 Establishing the Length

The FD must contain the RECORD LENGTH statement with the length coded as a symbol. The program must establish the length of the record to be written (1 to 32767 bytes) in this field before executing the WRITE.

If the length is not established explicitly by the program the length of the last record accessed will be used.

11.5.3 Successful Completion

Provided that no file condition or irrecoverable I/O error occurs WRITE will transfer the number of bytes given by the record length from A to the record identified by the key.

11.5.4 Programming Note

A WRITE may extend the file size if needed subject to the available space in the Unix file system.

11.6 The READ FIRST and READ LAST Statements

READ FIRST is used to read the very first record of the file. It is coded:

```
READ FIRST filename INTO A
```

Similarly READ LAST is used to read the very last record in the file. It is coded:

```
READ LAST filename INTO A
```

In both cases filename identifies the direct Unix file definition and A is a simple or indexed variable. If READ FIRST or READ LAST is attempted on an FD which is not open the program will be terminated in error.

11.6.1 Establishing the Length

The FD must contain the RECORD LENGTH statement with the length coded as a symbol. The program must establish the length of the record to be read (1 to 32767 bytes) in this field before executing a READ FIRST or READ LAST.

If the length is not established explicitly by the program the length of the last record accessed will be used.

11.6.2 File Conditions

A file boundary violation condition will be signalled if a READ FIRST or READ LAST attempts to input a record which is larger then the total file size, and which would therefore start or end outside the file extent.

11.6.3 Successful Completion

Provided that no file condition or irrecoverable I/O error occurs READ FIRST will set the key to address the first byte of the file (a value of zero), and READ LAST will set the key to be the file extent size less the length of the record to be processed. The record thus

identified will be transferred to A, the number of bytes transferred being given by the length field.

11.6.4 Programming Note

Use of READ FIRST is normally used to re-position at the start of the file prior to reading records sequentially using READ NEXT.

Use of READ LAST presupposes that the program has sufficient information about the file to be able to determine the length of the very last record (possibly because the length is fixed). It is important to note that it is the responsibility of the program to determine an appropriate record length for use by READ LAST, not that of the direct Unix access method itself.

11.7 The READ NEXT and READ PRIOR Statements

READ NEXT and READ PRIOR are used to process part or all of a file sequentially. READ NEXT is used to read the next sequential record, and it is coded:

```
READ NEXT filename INTO A
```

READ PRIOR is used to read the previous sequential record. It is coded:

```
READ PRIOR filename INTO A
```

In both cases filename identifies the direct Unix file definition and A is a simple or indexed variable. If a READ NEXT or READ PRIOR is attempted on an FD which is not already open the program will be terminated in error.

11.7.1 Establishing the Length

The FD must contain the RECORD LENGTH statement with the length coded as a symbol. The program must establish the length of the record to be read (1 to 32767 bytes) in this field before executing the READ NEXT or READ PRIOR.

If the length is not established explicitly by the program the length of the last record accessed will be used.

11.7.2 File Conditions

A file boundary violation condition will be signalled if READ NEXT or READ PRIOR attempts to input a record which is wholly or partially outside the file extent.

11.7.3 Successful Completion

Provided that no file condition or irrecoverable I/O error occurs READ NEXT will set the key to the number of the byte following the previous record accessed (if any), and READ PRIOR will set the key to the number of the byte record length bytes before the start of the previous record accessed. The record thus identified is transferred to A. In both cases the number of bytes transferred is given by the length field.

11.7.4 Programming Note

If a KEY statement was not specified in the file definition the FD still contains an internal key field which is set to 0 when the file

is opened. In this case READ NEXT can be used to read sequentially through the entire file; READ PRIOR at any point may be used to retrieve the preceding record; WRITE updates the last record thus retrieved; and READ rereads it. In addition READ FIRST and READ LAST may be used to position at either the start or end of the file.

When a KEY statement is specified a READ or WRITE can be used to position the file at any point. READ NEXT will continue sequential processing from the record following the one retrieved by READ or output by WRITE, and READ LAST will process sequentially from the record before it.

A WRITE NEXT statement sets the key field to the start of the record written, so that a subsequent READ NEXT statement will read the following record, and a READ PRIOR statement will read the previous record.

Note that when using READ PRIOR it is the responsibility of the program to determine the length of the record to be processed by some means (possibly because it has a fixed length), and not that of the direct Unix access method itself.

11.8 The READ Statement

READ is used to retrieve a record at random or reread the last record accessed. It is coded:

```
READ filename INTO A
```

Here filename identifies the direct Unix file definition and A is a simple or indexed variable. If READ is attempted on an FD which is not already open the program will be terminated in error.

11.8.1 Establishing the Key

If READ is to be used as a random access operation the FD must contain a KEY statement coded as:

```
KEY IS symbol
```

which causes:

```
02 symbol PIC 9(9) COMP
```

to be generated. The program must then move the byte number of the start of the record it is required to retrieve to this field before executing the READ statement.

If a KEY statement is not coded in the file definition the FD still contains an internal key field which is set to zero when the file is opened and incremented by READ NEXT and WRITE NEXT. In this case the only use of the READ statement is to reread the record previously retrieved by READ NEXT or written by WRITE NEXT.

11.8.2 Establishing the Length

The FD must contain the RECORD LENGTH statement with the length coded as a symbol. The program must establish the length of the record to be read (1 to 32767 bytes) in this field before executing the READ.

If the length is not established explicitly by the program the length of the last record accessed will be used.

11.8.3 File Conditions
A file boundary violation condition will be signalled if READ attempts to input a record which is wholly or partially outside the file extent.

11.8.4 Successful Completion
Providing no file condition or irrecoverable I/O error occurs, READ will transfer to A the number of bytes given by the record length of the record specified in the key field.

11.8.5 The READ PHYSICAL Statement
As the key of a DUAM file is a byte number, the READ PHYSICAL statement functions in exactly the same way as a READ.

11.9 The CLOSE Statement
CLOSE must be used to terminate the processing of a file. It is coded:

```
CLOSE filename [DELETE]
```

where filename identifies the direct Unix file definition.

11.9.1 Standard Processing
CLOSE always completes any outstanding I/O operations on the file and returns the FD to the status it possessed prior to being opened. Following CLOSE, the FD can be re-opened for the same (or a different) file by a subsequent OPEN NEW or OPEN OLD statement.

11.9.2 File Conditions
If the FD passed to the CLOSE was not open, then a file not open condition will be signalled.

11.9.3 Deletion
If a DELETE phrase is coded all the space the file occupies is returned to Unix file system and the file name is removed from the Unix directory. Following a CLOSE DELETE the file no longer exists.

11.9.4 Programming Notes
If you fail to close a direct Unix file then the Unix channel will remain open and will not be closed by any other System Manager process including exiting from System Manager. It is therefore very important to ensure that all direct Unix FD's are closed.

12. Direct MS-DOS Access Method

The direct DOS file organisation allows access to DOS files from the Global System Manager environment and may therefore only be run on Global System Manager DOS. The DOS file accessed is treated as a string of bytes numbered sequentially from zero upwards. Records are accessed by supplying the starting byte number as a key, together with the number of bytes to be read or written.

12.1 File Structure

12.1.1 Record Format

The format of the DOS file records is entirely under program control since you determine the length and starting byte of each record.

12.1.2 Specifying File Attributes

The attributes of a file, such as its unit-id, volume-id and file-id, specified in the file definition (FD) coded in the data division has no meaning but must be specified. It is recommended that the file-id is set to a descriptive variable name, the unit-id to "?" and the volume-id should not be set at all.

You must specify the name of the area which contains the record length. You can also define a key area to contain the starting byte number for use in random access READ and WRITE statements. Section 12.2 below describes those parts of the file definition which are specific to the basic direct file organisation, but the statements which are common to all organisations are defined in section 1.2.

12.1.3 File Processing Statements

A number of procedure division statements are provided to enable a DOS file to be processed. They are:

OPEN which must be executed prior to any other statement affecting the file;

READ to read a record at random;

READ FIRST to read the very first record in the file;

READ LAST to read the very last record in the file;

READ NEXT to read the next record during sequential processing;

READ PRIOR to read the previous record during sequential processing;

WRITE to update an existing record at random;

WRITE NEXT to write a record during sequential processing;

CLOSE to terminate processing of a file.

When any of the statements is executed a file condition, signalled by exception condition 2, may arise as explained in section 1.3.1. In addition, if OPTION ERROR or ON ERROR is specified in the FD, an

exception condition will be generated should an irrecoverable I/O error occur, as explained in section 1.6. Therefore it is usual to follow each file processing statement with an ON EXCEPTION statement. If you do not, and an exception condition arises, your program will be terminated in error.

12.1.4 Extent Boundary Checking

A write next pointer is not maintained for a direct DOS file, and you may use WRITE NEXT, WRITE, READ FIRST, READ LAST, READ NEXT, READ PRIOR and READ statements to access information anywhere within the allocated file extent. However, should any of these statements attempt to read a record which is partially or wholly outside the file extent, the file boundary violation file condition will be signalled.

12.1.5 The ORGANISATION Statement

If a program uses the basic direct access method then the statement:

```
ORGANISATION OR$98 TYPE 0 EXTENSION 128
```

must be coded in the data division before the first FD or data declaration.

12.2 The File Definition

The file definition for a direct DOS file is coded in either working storage or the linkage section as follows:

```
FD filename ORGANISATION OR$98
[ASSIGN TO UNIT "?" FILE file-id ]
[KEY IS keyname]
RECORD LENGTH IS length
[SIZE IS size]
[OPTION ERROR]
[ON ERROR intercept]
01 FILLER REDEFINES filename
02 FILLER PIC X(88)
02 PANAME PIC X(99) * DOS directory path and filename
02 FILLER PIC X
VALUE #00
```

The FD establishes a special group data item, 208 bytes in length, whose name is filename. The quantities file-id, DOS Directory path and filename, keyname, length and size appear as subordinate items within this group and can, if need be, be referred to by the application program.

It is possible to specify the DOS path and file name (which must be terminated with a #00 byte) before the program executes, In this case the quantity should be coded with a VALUE clause as a character string. If you can specify the size or length before the program executes, code the quantity as a numeric string. If the DOS directory path and filename is not Known until run-time then the quantity must be moved to the PANAME field during program execution. If the size and length is not Known until run-time symbol must be coded for the quantity. This symbol will then label a level 02 item which the user program is responsible for initialising.

12.2.1 The Filename

The filename is coded as a descriptive symbol. It serves to label the file definition, as explained in section 1.2.1.

12.2.2 The ORGANISATION Clause

The ORGANISATION clause must be coded as shown. It indicates that the basic direct access method is to be used.

12.2.3 The ASSIGN Statement

The ASSIGN statement is required in working storage and should be coded as shown, where file-id is only required as a descriptive symbol.

12.2.4 Optional Statement Placement

The KEY, RECORD LENGTH, SIZE, OPTION and ON ERROR statements may appear in any order following the ASSIGN statement. The RECORD LENGTH statement must be coded: the others are optional.

12.2.5 The KEY Statement

The KEY statement is only required if records are to be accessed at random. Keyname is specified as a symbol and the statement:

```
02 symbol PIC 9(9) COMP
```

is generated within the FD.

DDAM maintains the keyname field to contain the byte number of the start of the record last accessed. The byte number of the very first byte of the file counts as byte number zero.

The program must place the byte number of the start of the record it requires to access in the keyname field before executing a random READ or WRITE operation.

A successful READ NEXT or WRITE NEXT operation increments the keyname field by the length of the previous record accessed and then accesses the record thus identified: if the previous operation on the file was an OPEN then the first record is accessed.

A successful READ PRIOR decrements the keyname field by the length of the record to be read, and then accesses that record.

12.2.6 The RECORD LENGTH Statement

The RECORD LENGTH statement is always required. Length is specified as a symbol and the statement:

```
02 symbol PIC 9(4) COMP
```

is generated within the FD.

The program must place the length of the record to be read in the length field before executing a read or write statement. The field is set to zero when the file is opened, and is not altered by read and write operations.

12.2.7 The SIZE Statement

The SIZE statement is required only if you wish know the current size of the file. The file size will be maintained during processing.

12.2.8 The OPTION and ON ERROR Statements

OPTION ERROR should be coded only if you wish your program to regain control following an irrecoverable I/O error. ON ERROR should be coded if you wish to handle certain I/O errors specially. The processing of these statements is common to all file organisations, and is described in section 1.6.

12.3 The OPEN Statement

The OPEN statement is coded:

```
OPEN type filename
```

where type is the word NEW or OLD filename identifies the direct DOS file definition. An OPEN statement must be executed before any other operation affecting the file. If an OPEN is attempted but the FD is already open your program will be terminated in error.

OPEN NEW is used to create a new DOS file of the specified path and file name if one does not already exist or to truncate a file that already exists to zero length. OPEN OLD obtains exclusive access to an existing file.

12.3.1 File Conditions

When an OPEN OLD statement is executed the System Manager checks to see whether a file with the same DOS path and file name exists. If this is not the case, file not found is signalled.

12.3.2 Successful Completion

Providing no file condition or irrecoverable I/O error occurs an OPEN NEW results in an empty file of size zero being opened on with the specified DOS path and file name.

When an OPEN OLD statement completes successfully the file size is returned in the FD and can be accessed by the user program if a SIZE statement with the symbol option was coded.

The record length field in the FD is set to zero by a successful OPEN statement.

The keyname field in the FD is always set to zero following a Successful OPEN statement. This is to allow the file to be processed sequentially using a sequence of READ NEXT or WRITE NEXT statements as described below.

12.4 The WRITE NEXT Statement

WRITE NEXT is used to write all or part of a file sequentially. It is coded:

```
WRITE NEXT filename FROM A
```

Here filename identifies the Direct DOS file definition and A is a simple or indexed variable. If a WRITE NEXT is attempted on an FD which is not already open the program will be terminated in error.

12.4.1 Establishing the Length

The FD must contain the RECORD LENGTH statement with the length coded as a symbol. The program must establish the length of the record to be

written (1 to 32767 bytes) in this field before executing the WRITE NEXT.

If the length is not established explicitly by the program the length of the last record accessed will be used.

12.4.3 Successful Completion

Provided that no file condition or irrecoverable I/O error occurs WRITE NEXT will set the key to the number of the byte following the previous record accessed (if any) and transfer A to the record thus identified. The number of bytes transferred is given by the length field.

12.4.4 Programming Note

If a KEY statement was not specified in the file definition the FD still contains an internal key field which is set to zero when the file is opened. In this case WRITE NEXT can be used to write the entire file sequentially.

When a KEY statement is specified a READ or WRITE can be used to position the file at any point. WRITE NEXT will continue writing from the record following the one accessed by READ or WRITE.

The READ NEXT statement sets the key field to the record retrieved, so that a subsequent WRITE NEXT statement will write the following record.

The WRITE NEXT can cause the file size to be extended if needed, subject to space being available in the DOS file system.

12.5 The WRITE Statement

WRITE is used to write a record at random or rewrite the last record accessed. It is coded:

```
WRITE filename FROM A
```

Here filename identifies the direct DOS file definition and A is a simple or indexed variable. If a WRITE is attempted on an FD which is not already open the program will be terminated in error.

12.5.1 Establishing the Key

If WRITE is to be used as a random access operation the FD must contain a KEY statement coded as:

```
KEY IS symbol
```

which causes:

```
02 symbol PIC 9(9) COMP
```

to be generated. The program must then move the byte number of the start of the record it is required to retrieve to this field before executing the WRITE statement.

If a KEY statement is not coded in the file definition the FD still contains an internal key field which is set to zero when the file is opened and incremented by READ NEXT and WRITE NEXT. In this case the

only use of the WRITE statement is to rewrite the record previously retrieved by READ NEXT or written by WRITE NEXT.

12.5.2 Establishing the Length

The FD must contain the RECORD LENGTH statement with the length coded as a symbol. The program must establish the length of the record to be written (1 to 32767 bytes) in this field before executing the WRITE.

If the length is not established explicitly by the program the length of the last record accessed will be used.

12.5.3 Successful Completion

Provided that no file condition or irrecoverable I/O error occurs WRITE will transfer the number of bytes given by the record length from A to the record identified by the key.

12.5.4 Programming Note

A WRITE may extend the file size if needed subject to the available space in the DOS file system.

12.6 The READ FIRST and READ LAST Statements

READ FIRST is used to read the very first record of the file. It is coded:

```
READ FIRST filename INTO A
```

Similarly READ LAST is used to read the very last record in the file. It is coded:

```
READ LAST filename INTO A
```

In both cases filename identifies the direct DOS file definition and A is a simple or indexed variable. If READ FIRST or READ LAST is attempted on an FD which is not open the program will be terminated in error.

12.6.1 Establishing the Length

The FD must contain the RECORD LENGTH statement with the length coded as a symbol. The program must establish the length of the record to be read (1 to 32767 bytes) in this field before executing a READ FIRST or READ LAST.

If the length is not established explicitly by the program the length of the last record accessed will be used.

12.6.2 File Conditions

A file boundary violation condition will be signalled if a READ FIRST or READ LAST attempts to input a record which is larger then the total file size, and which would therefore start or end outside the file extent.

12.6.3 Successful Completion

Provided that no file condition or irrecoverable I/O error occurs READ FIRST will set the key to address the first byte of the file (a value of zero), and READ LAST will set the key to be the file extent size less the length of the record to be processed. The record thus

identified will be transferred to A, the number of bytes transferred being given by the length field.

12.6.4 Programming Note

Use of READ FIRST is normally used to re-position at the start of the file prior to reading records sequentially using READ NEXT.

Use of READ LAST presupposes that the program has sufficient information about the file to be able to determine the length of the very last record (possibly because the length is fixed). It is important to note that it is the responsibility of the program to determine an appropriate record length for use by READ LAST, not that of the direct DOS access method itself.

12.7 The READ NEXT and READ PRIOR Statements

READ NEXT and READ PRIOR are used to process part or all of a file sequentially. READ NEXT is used to read the next sequential record, and it is coded:

```
READ NEXT filename INTO A
```

READ PRIOR is used to read the previous sequential record. It is coded:

```
READ PRIOR filename INTO A
```

In both cases filename identifies the direct DOS file definition and A is a simple or indexed variable. If a READ NEXT or READ PRIOR is attempted on an FD which is not already open the program will be terminated in error.

12.7.1 Establishing the Length

The FD must contain the RECORD LENGTH statement with the length coded as a symbol. The program must establish the length of the record to be read (1 to 32767 bytes) in this field before executing the READ NEXT or READ PRIOR.

If the length is not established explicitly by the program the length of the last record accessed will be used.

12.7.2 File Conditions

A file boundary violation condition will be signalled if READ NEXT or READ PRIOR attempts to input a record which is wholly or partially outside the file extent.

12.7.3 Successful Completion

Provided that no file condition or irrecoverable I/O error occurs READ NEXT will set the key to the number of the byte following the previous record accessed (if any), and READ PRIOR will set the key to the number of the byte record length bytes before the start of the previous record accessed. The record thus identified is transferred to A. In both cases the number of bytes transferred is given by the length field.

12.7.4 Programming Note

If a KEY statement was not specified in the file definition the FD still contains an internal key field which is set to 0 when the file

is opened. In this case READ NEXT can be used to read sequentially through the entire file; READ PRIOR at any point may be used to retrieve the preceding record; WRITE updates the last record thus retrieved; and READ rereads it. In addition READ FIRST and READ LAST may be used to position at either the start or end of the file.

When a KEY statement is specified a READ or WRITE can be used to position the file at any point. READ NEXT will continue sequential processing from the record following the one retrieved by READ or output by WRITE, and READ LAST will process sequentially from the record before it.

A WRITE NEXT statement sets the key field to the start of the record written, so that a subsequent READ NEXT statement will read the following record, and a READ PRIOR statement will read the previous record.

Note that when using READ PRIOR it is the responsibility of the program to determine the length of the record to be processed by some means (possibly because it has a fixed length), and not that of the direct DOS access method itself.

12.8 The READ Statement

READ is used to retrieve a record at random or reread the last record accessed. It is coded:

```
READ filename INTO A
```

Here filename identifies the direct DOS file definition and A is a simple or indexed variable. If READ is attempted on an FD which is not already open the program will be terminated in error.

12.8.1 Establishing the Key

If READ is to be used as a random access operation the FD must contain a KEY statement coded as:

```
KEY IS symbol
```

which causes:

```
02 symbol PIC 9(9) COMP
```

to be generated. The program must then move the byte number of the start of the record it is required to retrieve to this field before executing the READ statement.

If a KEY statement is not coded in the file definition the FD still contains an internal key field which is set to zero when the file is opened and incremented by READ NEXT and WRITE NEXT. In this case the only use of the READ statement is to reread the record previously retrieved by READ NEXT or written by WRITE NEXT.

12.8.2 Establishing the Length

The FD must contain the RECORD LENGTH statement with the length coded as a symbol. The program must establish the length of the record to be read (1 to 32767 bytes) in this field before executing the READ.

If the length is not established explicitly by the program the length of the last record accessed will be used.

12.8.3 File Conditions

A file boundary violation condition will be signalled if READ attempts to input a record which is wholly or partially outside the file extent.

12.8.4 Successful Completion

Providing no file condition or irrecoverable I/O error occurs, READ will transfer to A the number of bytes given by the record length of the record specified in the key field.

12.8.5 The READ PHYSICAL Statement

As the key of a DDAM file is a byte number, the READ PHYSICAL statement functions in exactly the same way as a READ.

12.9 The CLOSE Statement

CLOSE must be used to terminate the processing of a file. It is coded:

```
CLOSE filename [DELETE]
```

where filename identifies the direct DOS file definition.

12.9.1 Standard Processing

CLOSE always completes any outstanding I/O operations on the file and returns the FD to the status it possessed prior to being opened. Following CLOSE, the FD can be re-opened for the same (or a different) file by a subsequent OPEN NEW or OPEN OLD statement.

12.9.2 File Conditions

If the FD passed to the CLOSE was not open, then a file not open condition will be signalled.

12.9.3 Deletion

If a DELETE phrase is coded all the space the file occupies is returned to DOS file system and the file name is removed from the DOS directory. Following a CLOSE DELETE the file no longer exists.

12.9.5 Programming Notes

If you fail to close a direct DOS file then the DOS channel will remain open and will not be closed by any other System Manager process including exiting from System Manager. It is therefore very important to ensure that all direct DOS FD's are closed.

13. The Direct Windows File Organisation

The direct Windows file organisation allows access to Windows files from the Global System Manager environment and may therefore only be run on Global System Manager Windows 95 or Windows-NT. The Windows file accessed is treated as a string of bytes numbered sequentially from zero upwards. Records are accessed by supplying the starting byte number as a key, together with the number of bytes to be read or written.

13.1 File Structure

13.1.1 Record Format

The format of the Windows file records is entirely under program control since you determine the length and starting byte of each record.

13.1.2 Specifying File Attributes

The attributes of a file, such as its unit-id, volume-id and file-id, specified in the file definition (FD) coded in the data division has no meaning but must be specified. It is recommended that the file-id is set to a descriptive variable name, the unit-id to "?" and the volume-id should not be set at all.

You must specify the name of the area which contains the record length. You can also define a key area to contain the starting byte number for use in random access READ and WRITE statements. Section 13.2 below describes those parts of the file definition which are specific to the basic direct file organisation, but the statements which are common to all organisations are defined in section 1.2.

13.1.3 File Processing Statements

A number of procedure division statements are provided to enable a Windows file to be processed. They are:

OPEN which must be executed prior to any other statement affecting the file;

READ to read a record at random;

READ FIRST to read the very first record in the file;

READ LAST to read the very last record in the file;

READ NEXT to read the next record during sequential processing;

READ PRIOR to read the previous record during sequential processing;

WRITE to update an existing record at random;

WRITE NEXT to write a record during sequential processing;

CLOSE to terminate processing of a file.

When any of the statements is executed a file condition, signalled by exception condition 2, may arise as explained in section 1.3.1. In addition, if OPTION ERROR or ON ERROR is specified in the FD, an exception condition will be generated should an irrecoverable I/O error occur, as explained in section 1.6. Therefore it is usual to follow each file processing statement with an ON EXCEPTION statement. If you do not, and an exception condition arises, your program will be terminated in error.

13.1.4 Extent Boundary Checking

A write next pointer is not maintained for a direct Windows file, and you may use WRITE NEXT, WRITE, READ FIRST, READ LAST, READ NEXT, READ PRIOR and READ statements to access information anywhere within the allocated file extent. However, should any of these statements attempt to read a record which is partially or wholly outside the file extent, the file boundary violation file condition will be signalled.

13.1.5 The ORGANISATION Statement

If a program uses the basic direct access method then the statement:

```
ORGANISATION OR$98W TYPE 0 EXTENSION 128
```

must be coded in the data division before the first FD or data declaration.

13.2 The File Definition

The file definition for a direct Windows file is coded in either working storage or the linkage section as follows:

```
FD filename ORGANISATION OR$98W
[ASSIGN TO UNIT "?" FILE file-id ]
[KEY IS keyname]
RECORD LENGTH IS length
[SIZE IS size]
[OPTION ERROR]
[ON ERROR intercept]
01 FILLER REDEFINES filename
02 FILLER PIC X(88)
02 PANAME PIC X(99) * Windows directory path and filename
02 FILLER PIC X
VALUE #00
```

The FD establishes a special group data item, 208 bytes in length, whose name is filename. The quantities file-id, Windows Directory path and filename, keyname, length and size appear as subordinate items within this group and can, if need be, be referred to by the application program.

It is possible to specify the Windows path and file name (which must be terminated with a #00 byte) before the program executes, In this case the quantity should be coded with a VALUE clause as a character string. If you can specify the size or length before the program executes, code the quantity as a numeric string. If the Windows directory path and filename is not known until run-time then the quantity must be moved to the PANAME field during program execution. If the size and length is not known until run-time symbol must be coded for the quantity. This symbol will then label a level 02 item which the user program is responsible for initialising.

13.2.1 The Filename
The filename is coded as a descriptive symbol. It serves to label the file definition, as explained in section 1.2.1.

13.2.2 The ORGANISATION Clause
The ORGANISATION clause must be coded as shown. It indicates that the Windows direct access method is to be used.

13.2.3 The ASSIGN Statement
The ASSIGN statement is required in working storage and should be coded as shown, where file-id is only required as a descriptive symbol.

13.2.4 Optional Statement Placement
The KEY, RECORD LENGTH, SIZE, OPTION and ON ERROR statements may appear in any order following the ASSIGN statement. The RECORD LENGTH statement must be coded: the others are optional.

13.2.5 The KEY Statement
The KEY statement is only required if records are to be accessed at random. Keyname is specified as a symbol and the statement:

```
02 symbol PIC 9(9) COMP
```

is generated within the FD.

DWAM maintains the keyname field to contain the byte number of the start of the record last accessed. The byte number of the very first byte of the file counts as byte number zero.

The program must place the byte number of the start of the record it requires to access in the keyname field before executing a random READ or WRITE operation.

A successful READ NEXT or WRITE NEXT operation increments the keyname field by the length of the previous record accessed and then accesses the record thus identified: if the previous operation on the file was an OPEN then the first record is accessed.

A successful READ PRIOR decrements the keyname field by the length of the record to be read, and then accesses that record.

13.2.6 The RECORD LENGTH Statement
The RECORD LENGTH statement is always required. Length is specified as a symbol and the statement:

```
02 symbol PIC 9(4) COMP
```

is generated within the FD.

The program must place the length of the record to be read in the length field before executing a read or write statement. The field is set to zero when the file is opened, and is not altered by read and write operations.

13.2.7 The SIZE Statement

The SIZE statement is required only if you wish know the current size of the file. The file size will be maintained during processing.

13.2.8 The OPTION and ON ERROR Statements

OPTION ERROR should be coded only if you wish your program to regain control following an irrecoverable I/O error. ON ERROR should be coded if you wish to handle certain I/O errors specially. The processing of these statements is common to all file organisations, and is described in section 1.6.

13.3 The OPEN Statement

The OPEN statement is coded:

```
OPEN type filename
```

where type is the word NEW or OLD filename identifies the direct Windows file definition. An OPEN statement must be executed before any other operation affecting the file. If an OPEN is attempted but the FD is already open your program will be terminated in error.

OPEN NEW is used to create a new Windows file of the specified path and file name if one does not already exist or to truncate a file that already exists to zero length. OPEN OLD obtains exclusive access to an existing file.

13.3.1 File Conditions

When an OPEN OLD statement is executed the System Manager checks to see whether a file with the same Windows path and file name exists. If this is not the case, file not found is signalled.

13.3.2 Successful Completion

Providing no file condition or irrecoverable I/O error occurs an OPEN NEW results in an empty file of size zero being opened on with the specified Windows path and file name.

When an OPEN OLD statement completes successfully the file size is returned in the FD and can be accessed by the user program if a SIZE statement with the symbol option was coded.

The record length field in the FD is set to zero by a successful OPEN statement.

The keyname field in the FD is always set to zero following a successful OPEN statement. This is to allow the file to be processed sequentially using a sequence of READ NEXT or WRITE NEXT statements as described below.

13.4 The WRITE NEXT Statement

WRITE NEXT is used to write all or part of a file sequentially. It is coded:

```
WRITE NEXT filename FROM A
```

Here filename identifies the Direct Windows file definition and A is a simple or indexed variable. If a WRITE NEXT is attempted on an FD which is not already open the program will be terminated in error.

13.4.1 Establishing the Length

The FD must contain the RECORD LENGTH statement with the length coded as a symbol. The program must establish the length of the record to be written (1 to 32767 bytes) in this field before executing the WRITE NEXT.

If the length is not established explicitly by the program the length of the last record accessed will be used.

13.4.3 Successful Completion

Provided that no file condition or irrecoverable I/O error occurs WRITE NEXT will set the key to the number of the byte following the previous record accessed (if any) and transfer A to the record thus identified. The number of bytes transferred is given by the length field.

13.4.4 Programming Note

If a KEY statement was not specified in the file definition the FD still contains an internal key field which is set to zero when the file is opened. In this case WRITE NEXT can be used to write the entire file sequentially.

When a KEY statement is specified a READ or WRITE can be used to position the file at any point. WRITE NEXT will continue writing from the record following the one accessed by READ or WRITE.

The READ NEXT statement sets the key field to the record retrieved, so that a subsequent WRITE NEXT statement will write the following record.

The WRITE NEXT can cause the file size to be extended if needed, subject to space being available in the Windows file system.

13.5 The WRITE Statement

WRITE is used to write a record at random or rewrite the last record accessed. It is coded:

```
WRITE filename FROM A
```

Here filename identifies the direct Windows file definition and A is a simple or indexed variable. If a WRITE is attempted on an FD which is not already open the program will be terminated in error.

13.5.1 Establishing the Key

If WRITE is to be used as a random access operation the FD must contain a KEY statement coded as:

```
KEY IS symbol
```

which causes:

```
02 symbol PIC 9(9) COMP
```

to be generated. The program must then move the byte number of the start of the record it is required to retrieve to this field before executing the WRITE statement.

If a KEY statement is not coded in the file definition the FD still contains an internal key field which is set to zero when the file is opened and incremented by READ NEXT and WRITE NEXT. In this case the only use of the WRITE statement is to rewrite the record previously retrieved by READ NEXT or written by WRITE NEXT.

13.5.2 Establishing the Length

The FD must contain the RECORD LENGTH statement with the length coded as a symbol. The program must establish the length of the record to be written (1 to 32767 bytes) in this field before executing the WRITE.

If the length is not established explicitly by the program the length of the last record accessed will be used.

13.5.3 Successful Completion

Provided that no file condition or irrecoverable I/O error occurs WRITE will transfer the number of bytes given by the record length from A to the record identified by the key.

13.5.4 Programming Note

A WRITE may extend the file size if needed subject to the available space in the Windows file system.

13.6 The READ FIRST and READ LAST Statements

READ FIRST is used to read the very first record of the file. It is coded:

```
READ FIRST filename INTO A
```

Similarly READ LAST is used to read the very last record in the file. It is coded:

```
READ LAST filename INTO A
```

In both cases filename identifies the direct Windows file definition and A is a simple or indexed variable. If READ FIRST or READ LAST is attempted on an FD which is not open the program will be terminated in error.

13.6.1 Establishing the Length

The FD must contain the RECORD LENGTH statement with the length coded as a symbol. The program must establish the length of the record to be read (1 to 32767 bytes) in this field before executing a READ FIRST or READ LAST.

If the length is not established explicitly by the program the length of the last record accessed will be used.

13.6.2 File Conditions

A file boundary violation condition will be signalled if a READ FIRST or READ LAST attempts to input a record which is larger then the total file size, and which would therefore start or end outside the file extent.

13.6.3 Successful Completion

Provided that no file condition or irrecoverable I/O error occurs READ FIRST will set the key to address the first byte of the file (a value

of zero), and READ LAST will set the key to be the file extent size less the length of the record to be processed. The record thus identified will be transferred to A, the number of bytes transferred being given by the length field.

13.6.4 Programming Note

Use of READ FIRST is normally used to re-position at the start of the file prior to reading records sequentially using READ NEXT.

Use of READ LAST presupposes that the program has sufficient information about the file to be able to determine the length of the very last record (possibly because the length is fixed). It is important to note that it is the responsibility of the program to determine an appropriate record length for use by READ LAST, not that of the direct Windows access method itself.

13.7 The READ NEXT and READ PRIOR Statements

READ NEXT and READ PRIOR are used to process part or all of a file sequentially. READ NEXT is used to read the next sequential record, and it is coded:

```
READ NEXT filename INTO A
```

READ PRIOR is used to read the previous sequential record. It is coded:

```
READ PRIOR filename INTO A
```

In both cases filename identifies the direct Windows file definition and A is a simple or indexed variable. If a READ NEXT or READ PRIOR is attempted on an FD which is not already open the program will be terminated in error.

13.7.1 Establishing the Length

The FD must contain the RECORD LENGTH statement with the length coded as a symbol. The program must establish the length of the record to be read (1 to 32767 bytes) in this field before executing the READ NEXT or READ PRIOR.

If the length is not established explicitly by the program the length of the last record accessed will be used.

13.7.2 File Conditions

A file boundary violation condition will be signalled if READ NEXT or READ PRIOR attempts to input a record which is wholly or partially outside the file extent.

13.7.3 Successful Completion

Provided that no file condition or irrecoverable I/O error occurs READ NEXT will set the key to the number of the byte following the previous record accessed (if any), and READ PRIOR will set the key to the number of the byte record length bytes before the start of the previous record accessed. The record thus identified is transferred to A. In both cases the number of bytes transferred is given by the length field.

13.7.4 Programming Note

If a KEY statement was not specified in the file definition the FD still contains an internal key field which is set to 0 when the file is opened. In this case READ NEXT can be used to read sequentially through the entire file; READ PRIOR at any point may be used to retrieve the preceding record; WRITE updates the last record thus retrieved; and READ rereads it. In addition READ FIRST and READ LAST may be used to position at either the start or end of the file.

When a KEY statement is specified a READ or WRITE can be used to position the file at any point. READ NEXT will continue sequential processing from the record following the one retrieved by READ or output by WRITE, and READ LAST will process sequentially from the record before it.

A WRITE NEXT statement sets the key field to the start of the record written, so that a subsequent READ NEXT statement will read the following record, and a READ PRIOR statement will read the previous record.

Note that when using READ PRIOR it is the responsibility of the program to determine the length of the record to be processed by some means (possibly because it has a fixed length), and not that of the direct Windows access method itself.

13.8 The READ Statement

READ is used to retrieve a record at random or reread the last record accessed. It is coded:

```
READ filename INTO A
```

Here filename identifies the direct Windows file definition and A is a simple or indexed variable. If READ is attempted on an FD which is not already open the program will be terminated in error.

13.8.1 Establishing the Key

If READ is to be used as a random access operation the FD must contain a KEY statement coded as:

```
KEY IS symbol
```

which causes:

```
02 symbol PIC 9(9) COMP
```

to be generated. The program must then move the byte number of the start of the record it is required to retrieve to this field before executing the READ statement.

If a KEY statement is not coded in the file definition the FD still contains an internal key field which is set to zero when the file is opened and incremented by READ NEXT and WRITE NEXT. In this case the only use of the READ statement is to reread the record previously retrieved by READ NEXT or written by WRITE NEXT.

13.8.2 Establishing the Length

The FD must contain the RECORD LENGTH statement with the length coded as a symbol. The program must establish the length of the record to be read (1 to 32767 bytes) in this field before executing the READ.

If the length is not established explicitly by the program the length of the last record accessed will be used.

13.8.3 File Conditions
A file boundary violation condition will be signalled if READ attempts to input a record which is wholly or partially outside the file extent.

13.8.4 Successful Completion
Providing no file condition or irrecoverable I/O error occurs, READ will transfer to A the number of bytes given by the record length of the record specified in the key field.

13.8.5 The READ PHYSICAL Statement
As the key of a DDAM file is a byte number, the READ PHYSICAL statement functions in exactly the same way as a READ.

13.9 The CLOSE Statement
CLOSE must be used to terminate the processing of a file. It is coded:

```
CLOSE filename [DELETE]
```

where filename identifies the direct Windows file definition.

13.9.1 Standard Processing
CLOSE always completes any outstanding I/O operations on the file and returns the FD to the status it possessed prior to being opened. Following CLOSE, the FD can be re-opened for the same (or a different) file by a subsequent OPEN NEW or OPEN OLD statement.

13.9.2 File Conditions
If the FD passed to the CLOSE was not open, then a file not open condition will be signalled.

13.9.3 Deletion
If a DELETE phrase is coded all the space the file occupies is returned to Windows file system and the file name is removed from the Windows directory. Following a CLOSE DELETE the file no longer exists.

13.9.5 Programming Notes
If you fail to close a direct Windows file then the Windows channel will remain open and will not be closed by any other System Manager process including exiting from System Manager. It is therefore very important to ensure that all direct Windows FD's are closed.

14. The Open Direct File Organisation

The Open Direct file organisation allows access to DOS, Unix and Windows files from the Global System Manager environment and may therefore only be run only on Global System Manager DOS, Unix, Windows 95 and Windows-NT. The file accessed is treated as a string of bytes numbered sequentially from zero upwards. Records are accessed by supplying the starting byte number as key, together with the number of bytes to be read or written.

This access method is only available under V8.1 (revision f) or later System Manager.

14.1 File Structure

14.1.1 Record Format

The format of the file records is entirely under program control since you determine the length and starting byte of each record.

14.1.2 Specifying File Attributes

The attributes of a file, such as its unit-id, volume-id and file-id, specified in the file definition (FD) coded in the data division has no meaning but must be specified. It is recommended that the file-id is set to a descriptive variable name, the unit-id to "?" and the volume-id should not be set at all.

You must specify the name of the area which contains the record length. You can also define a key area to contain the starting byte number for use in random access READ and WRITE statements. Section 14.2 below describes those parts of the file definition which are specific to the open direct file organisation, but the statements which are common to all organisations are defined in section 1.2.

14.1.3 File Processing Statements

A number of procedure division statements are provided to enable a file to be processed. They are:

OPEN which must be executed prior to any other statement affecting the file;

READ to read a record at random;

READ FIRST to read the very first record in the file;

READ LAST to read the very last record in the file;

READ NEXT to read the next record during sequential processing;

READ PRIOR to read the previous record during sequential processing;

WRITE to update an existing record at random;

WRITE NEXT to write a record during sequential processing;

CLOSE to terminate processing of a file.

When any of the statements is executed a file condition, signalled by exception condition 2, may arise as explained in section 1.3.1. In addition, if OPTION ERROR or ON ERROR is specified in the FD, an exception condition will be generated should an irrecoverable I/O error occur, as explained in section 1.6. Therefore it is usual to follow each file processing statement with an ON EXCEPTION statement. If you do not, and an exception condition arises, your program will be terminated in error.

14.1.4 Extent Boundary Checking

A write next pointer is not maintained for a direct file, and you may use WRITE NEXT, WRITE, READ FIRST, READ LAST, READ NEXT, READ PRIOR and READ statements to access information anywhere within the allocated file extent. However, should any of these statements attempt to read or write a record which is partially or wholly outside the file extent, the file boundary violation file condition will be signalled.

14.1.5 The ORGANISATION Statement

If a program uses the basic direct access method then the statement:

```
ORGANISATION OR$98 TYPE 0 EXTENSION 128
```

must be coded in the data division before the first FD or data declaration.

14.1.6 Programming Notes

The open direct access method is a pageable routine (See System Subroutines manual for details and its interface routine must be linked specially. You must include the following line in the $link dialogue before linking either C.$PAGES, C.$APF or C.$MCOB:

```
$44 LINK:C.$PAGES/AG$Z UNIT:$S
```

The routine will appear in the link map with program name AG$Z. Note that if you do not specifically link this routine then the direct DOS access method (AC$A) will be included instead.

14.2 The File Definition

The file definition for a direct file is coded in either working storage or the linkage section as follows:

```
FD filename ORGANISATION OR$98
[ASSIGN TO UNIT "?" FILE file-id ]
[KEY IS keyname]
RECORD LENGTH IS length
[SIZE IS size]
[OPTION ERROR]
[ON ERROR intercept]
01 FILLER REDEFINES filename
02 FILLER PIC X(88)
02 PANAME PIC X(99) * directory path and filename
02 FILLER PIC X
VALUE #00
02 FILLER PIC X(8)
02 PAPERM PIC 9(4) COMP * Unix permissions
```

The FD establishes a special group data item, 208 bytes in length, whose name is filename. The quantities file-id, Directory path and filename, keyname, length and size appear as subordinate items within this group and can, if need be, be referred to by the application program.

It is possible to specify the path and file name (which must be terminated with a #00 byte) before the program executes, In this case the quantity should be coded with a VALUE clause as a character string. If you can specify the size or length before the program executes, code the quantity as a numeric string. If the directory path and filename is not known until run-time then the quantity must be moved to the PANAME field during program execution. If the size or length is not known until run-time a symbol must be coded for the quantity. This symbol will then label a level 02 item which the user program is responsible for initialising. The permissions field is only of value for Unix files and allows you to set the Unix permissions before opening a file.

14.2.1 The Filename

The filename is coded as a descriptive symbol. It serves to label the file definition, as explained in section 1.2.1.

14.2.2 The ORGANISATION Clause

The ORGANISATION clause must be coded as shown. It indicates that the open direct access method is to be used.

14.2.3 The ASSIGN Statement

The ASSIGN statement is required in working storage and should be coded as shown, where file-id is only required as a descriptive symbol.

14.2.4 Optional Statement Placement

The KEY, RECORD LENGTH, SIZE, OPTION and ON ERROR statements may appear in any order following the ASSIGN statement. The RECORD LENGTH statement must be coded: the others are optional.

14.2.5 The KEY Statement

The KEY statement is only required if records are to be accessed at random. Keyname is specified as a symbol and the statement:

```
02 symbol PIC 9(9) COMP
```

is generated within the FD.

ODAM maintains the keyname field to contain the byte number of the start of the record last accessed. The byte number of the very first byte of the file counts as byte number zero.

The program must place the byte number of the start of the record it requires to access in the keyname field before executing a random READ or WRITE operation.

A successful READ NEXT or WRITE NEXT operation increments the keyname field by the length of the previous record accessed and then accesses the record thus identified: if the previous operation on the file was an OPEN then the first record is accessed.

A successful READ PRIOR decrements the keyname field by the length of the record to be read, and then accesses that record.

14.2.6 The RECORD LENGTH Statement

The RECORD LENGTH statement is always required. Length is specified as a symbol and the statement:

```
02 symbol PIC 9(4) COMP
```

is generated within the FD.

The program must place the length of the record to be read in the length field before executing a read or write statement. The field is set to zero when the file is opened, and is not altered by read and write operations.

14.2.7 The SIZE Statement

The SIZE statement is required only if you wish know the current size of the file. The file size will be maintained during processing.

14.2.8 The OPTION and ON ERROR Statements

OPTION ERROR should be coded only if you wish your program to regain control following an irrecoverable I/O error. ON ERROR should be coded if you wish to handle certain I/O errors specially. The processing of these statements is common to all file organisations, and is described in section 1.6.

14.3 The OPEN Statement

The OPEN statement is coded:

```
OPEN type filename
```

where type is the word NEW or OLD filename identifies the direct file definition. An OPEN statement must be executed before any other operation affecting the file. If an OPEN is attempted but the FD is already open your program will be terminated in error.

OPEN NEW is used to create a new file of the specified path and file name if one does not already exist or to truncate a file that already exists to zero length. If there is no existing file the new file will be created. For Unix files the new file will be created with Unix permissions 600 octal (i.e. RW access for the owner) unless otherwise specified by the PAPERM field. OPEN OLD obtains exclusive access to an existing file.

14.3.1 File Conditions

When an OPEN OLD statement is executed the System Manager checks to see whether a file with the same path and file name exists. If this is not the case, file not found is signalled.

14.3.2 Successful Completion

Providing no file condition or irrecoverable I/O error occurs an OPEN NEW results in an empty file of size zero being opened on with the specified path and file name.

When an OPEN OLD statement completes successfully the file size is returned in the FD and can be accessed by the user program if a SIZE statement with the symbol option was coded.

The record length field in the FD is set to zero by a successful OPEN statement.

The keyname field in the FD is always set to zero following a successful OPEN statement. This is to allow the file to be processed sequentially using a sequence of READ NEXT or WRITE NEXT statements as described below.

14.4 The WRITE NEXT Statement

WRITE NEXT is used to write all or part of a file sequentially. It is coded:

```
WRITE NEXT filename FROM A
```

Here filename identifies the Direct file definition and A is a simple or indexed variable. If a WRITE NEXT is attempted on an FD which is not already open the program will be terminated in error.

14.4.1 Establishing the Length

The FD must contain the RECORD LENGTH statement with the length coded as a symbol. The program must establish the length of the record to be written (1 to 32767 bytes) in this field before executing the WRITE NEXT.

If the length is not established explicitly by the program the length of the last record accessed will be used.

14.4.3 Successful Completion

Provided that no file condition or irrecoverable I/O error occurs WRITE NEXT will set the key to the number of the byte following the previous record accessed (if any) and transfer A to the record thus identified. The number of bytes transferred is given by the length field.

14.4.4 Programming Note

If a KEY statement was not specified in the file definition the FD still contains an internal key field which is set to zero when the file is opened. In this case WRITE NEXT can be used to write the entire file sequentially.

When a KEY statement is specified a READ or WRITE can be used to position the file at any point. WRITE NEXT will continue writing from the record following the one accessed by READ or WRITE.

The READ NEXT statement sets the key field to the record retrieved, so that a subsequent WRITE NEXT statement will write the following record.

The WRITE NEXT can cause the file size to be extended if needed, subject to space being available in the file system.

14.5 The WRITE Statement

WRITE is used to write a record at random or rewrite the last record accessed. It is coded:

```
WRITE filename FROM A
```

Here filename identifies the direct file definition and A is a simple or indexed variable. If a WRITE is attempted on an FD which is not already open the program will be terminated in error.

14.5.1 Establishing the Key

If WRITE is to be used as a random access operation the FD must contain a KEY statement coded as:

```
KEY IS symbol
```

which causes:

```
02 symbol PIC 9(9) COMP
```

to be generated. The program must then move the byte number of the start of the record it is required to retrieve to this field before executing the WRITE statement.

If a KEY statement is not coded in the file definition the FD still contains an internal key field which is set to zero when the file is opened and incremented by READ NEXT and WRITE NEXT. In this case the only use of the WRITE statement is to rewrite the record previously retrieved by READ NEXT or written by WRITE NEXT.

14.5.2 Establishing the Length

The FD must contain the RECORD LENGTH statement with the length coded as a symbol. The program must establish the length of the record to be written (1 to 32767 bytes) in this field before executing the WRITE.

If the length is not established explicitly by the program the length of the last record accessed will be used.

14.5.3 Successful Completion

Provided that no file condition or irrecoverable I/O error occurs WRITE will transfer the number of bytes given by the record length from A to the record identified by the key.

14.5.4 Programming Note

A WRITE may extend the file size if needed subject to the available space in the file system.

14.6 The READ FIRST and READ LAST Statements

READ FIRST is used to read the very first record of the file. It is coded:

```
READ FIRST filename INTO A
```

Similarly READ LAST is used to read the very last record in the file. It is coded:

```
READ LAST filename INTO A
```

In both cases filename identifies the direct file definition and A is a simple or indexed variable. If READ FIRST or READ LAST is attempted on an FD which is not open the program will be terminated in error.

14.6.1 Establishing the Length

The FD must contain the RECORD LENGTH statement with the length coded as a symbol. The program must establish the length of the record to be read (1 to 32767 bytes) in this field before executing a READ FIRST or READ LAST.

If the length is not established explicitly by the program the length of the last record accessed will be used.

14.6.2 File Conditions

A file boundary violation condition will be signalled if a READFIRST or READ LAST attempts to input a record which is larger then the total file size, and which would therefore start or end outside the file extent.

14.6.3 Successful Completion

Provided that no file condition or irrecoverable I/O error occurs READ FIRST will set the key to address the first byte of the file (a value of zero), and READ LAST will set the key to be the file extent size less the length of the record to be processed. The record thus identified will be transferred to A, the number of bytes transferred being given by the length field.

14.6.4 Programming Note

Use of READ FIRST is normally used to re-position at the start of the file prior to reading records sequentially using READ NEXT.

Use of READ LAST presupposes that the program has sufficient information about the file to be able to determine the length of the very last record (possibly because the length is fixed). It is important to note that it is the responsibility of the program to determine an appropriate record length for use by READ LAST, not that of the direct access method itself.

14.7 The READ NEXT and READ PRIOR Statements

READ NEXT and READ PRIOR are used to process part or all of a file sequentially. READ NEXT is used to read the next sequential record, and it is coded:

```
READ NEXT filename INTO A
```

READ PRIOR is used to read the previous sequential record. It is coded:

```
READ PRIOR filename INTO A
```

In both cases filename identifies the direct file definition and A is a simple or indexed variable. If a READ NEXT or READ PRIOR is attempted on an FD which is not already open the program will be terminated in error.

14.7.1 Establishing the Length

The FD must contain the RECORD LENGTH statement with the length coded as a symbol. The program must establish the length of the record to be read (1 to 32767 bytes) in this field before executing the READ NEXT or READ PRIOR.

If the length is not established explicitly by the program the length of the last record accessed will be used.

14.7.2 File Conditions

A file boundary violation condition will be signalled if READ NEXT or READ PRIOR attempts to input a record which is wholly or partially outside the file extent.

14.7.3 Successful Completion

Provided that no file condition or irrecoverable I/O error occurs READ NEXT will set the key to the number of the byte following the previous record accessed (if any), and READ PRIOR will set the key to the number of the byte record length bytes before the start of the previous record accessed. The record thus identified is transferred to A. In both cases the number of bytes transferred is given by the length field.

14.7.4 Programming Note

If a KEY statement was not specified in the file definition the FD still contains an internal key field which is set to 0 when the file is opened. In this case READ NEXT can be used to read sequentially through the entire file; READ PRIOR at any point may be used to retrieve the preceding record; WRITE updates the last record thus retrieved; and READ rereads it. In addition READ FIRST and READ LAST may be used to position at either the start or end of the file.

When a KEY statement is specified a READ or WRITE can be used to position the file at any point. READ NEXT will continue sequential processing from the record following the one retrieved by READ or output by WRITE, and READ LAST will process sequentially from the record before it.

A WRITE NEXT statement sets the key field to the start of the record written, so that a subsequent READ NEXT statement will read the following record, and a READ PRIOR statement will read the previous record.

Note that when using READ PRIOR it is the responsibility of the program to determine the length of the record to be processed by some means (possibly because it has a fixed length), and not that of the direct access method itself.

14.8 The READ Statement

READ is used to retrieve a record at random or reread the last record accessed. It is coded:

```
READ filename INTO A
```

Here filename identifies the direct file definition and A is a simple or indexed variable. If READ is attempted on an FD which is not already open the program will be terminated in error.

14.8.1 Establishing the Key

If READ is to be used as a random access operation the FD must contain a KEY statement coded as:

```
KEY IS symbol
```

which causes:

```
02 symbol PIC 9(9) COMP
```

to be generated. The program must then move the byte number of the start of the record it is required to retrieve to this field before executing the READ statement.

If a KEY statement is not coded in the file definition the FD still contains an internal key field which is set to zero when the file is opened and incremented by READ NEXT and WRITE NEXT. In this case the only use of the READ statement is to reread the record previously retrieved by READ NEXT or written by WRITE NEXT.

14.8.2 Establishing the Length

The FD must contain the RECORD LENGTH statement with the length coded as a symbol. The program must establish the length of the record to be read (1 to 32767 bytes) in this field before executing the READ.

If the length is not established explicitly by the program the length of the last record accessed will be used.

14.8.3 File Conditions

A file boundary violation condition will be signalled if READ attempts to input a record which is wholly or partially outside the file extent.

14.8.4 Successful Completion

Providing no file condition or irrecoverable I/O error occurs, READ will transfer to A the number of bytes given by the record length of the record specified in the key field.

14.8.5 The READ PHYSICAL Statement

As the key of a ODAM file is a byte number, the READ PHYSICAL statement functions in exactly the same way as a READ.

14.9 The CLOSE Statement

CLOSE must be used to terminate the processing of a file. It is coded:

```
CLOSE filename [DELETE]
```

where filename identifies the direct file definition.

14.9.1 Standard Processing

CLOSE always completes any outstanding I/O operations on the file and returns the FD to the status it possessed prior to being opened. Following CLOSE, the FD can be re-opened for the same (or a different) file by a subsequent OPEN NEW or OPEN OLD statement.

14.9.2 File Conditions

If the FD passed to the CLOSE was not open, then a file not open condition will be signalled.

14.9.3 Deletion

If a DELETE phrase is coded all the space the file occupies is returned to file system and the file name is removed from the directory. Following a CLOSE DELETE the file no longer exists.

14.9.5 Programming Notes

If you fail to close a direct file then the channel will remain open and will not be closed by any other System Manager process including exiting from System Manager. It is therefore very important to ensure that all direct FD's are closed.

15. File Management Routines

15.1 The File Conversion Routine, CONV$

The CONV$ system routine reads an input file, performs a conversion process, and creates an output file. The type of conversion is governed by the ORGANISATION clauses in the FD statements for the input and output files. The routine can be employed to create an indexed sequential file from a relative sequential file; to produce an empty indexed sequential file; to extract a relative sequential file from an indexed sequential file; to create a reorganised indexed sequential file from an existing indexed sequential file; and to produce a relative sequential file free of logically deleted records from an existing relative sequential file.

An indexed sequential file processed by the routine must occupy direct access storage.

The records of the input file must begin with a two-byte record type code, as described in 2.1.1 and 3.1.1. Records whose first byte is an asterisk character are considered to be logically deleted and are not transferred to the output file, unless the record type is *!, as explained in 3.1.9.

If either the input or the output file is indexed sequential the records must contain a two-byte link field and a record key following the type code. See 3.1.1.

15.1.1 Invocation

The file conversion routine is invoked by a CALL of the form:

```
CALL CONV$ USING [filename-i] filename-o area
```

Here filename-i is the name of the file definition for the input file; filename-o is the name of the file definition for the output file; and area is the name of a data area, long enough to hold a single record of the input file, which CONV$ will use as a work area. The input file may be omitted only when creating an empty indexed sequential file.

The file definitions associated with filename-i and filename-o must both be closed when CONV$ is called and they will remain closed when the routine returns control. If volume identification checking is specified in either or both FD's it will be applied in the normal way when the conversion routine opens the files for processing.

If a file with the same name as the output file already exists on the output volume it will be deleted. The output file is always allocated anew. It is allocated the extent specified in its file definition's SIZE statement except that, if the SIZE statement is omitted, or the actual size is specified as zero, the input file extent size will be used.

If the size specified is 999999999 the output file extent will be allocated the maximum amount of contiguous free space available.

The ISAM file size calculation routine, CALC$, described in Section 15.13 can be used to determine the extent required for an indexed sequential file.

15.1.2 Exceptions

Exception condition 1 is signalled if an irrecoverable I/O error arises on either the input or output file.

Exception condition 2 may be generated for a variety of reasons, each of which is identified by the value of $$RES:

- Input file is not of indicated organisation ($$RES="1");
- Keys out of sequence when creating an indexed sequential file from a relative sequential file ($$RES="2");
- Input file not found ($$RES="3").
- Output file capacity exceeded ($$RES="5").

When control is returned following an exception, both FD's will be closed in the normal way. An incomplete output file may be in existence in this case. If so it can be deleted either by using the file utility command program interactively or by using the conversion routine again.

15.1.3 Creating an Indexed Sequential File

An indexed sequential file is created by constructing a relative sequential file containing standard format records in ascending key sequence and then using CONV$ to convert the file to indexed sequential format.

The input file will be the relative sequential file previously created. Its file definition requires only an FD and an ASSIGN statement.

The output file is the new indexed sequential file to be produced by the conversion process. In its file definition you must specify that the file organisation is indexed sequential, and you must provide an ASSIGN statement. The RECORD LENGTH is unnecessary: it is taken from the input file. Unless you supply a SIZE statement and establish a non-zero size the file will be allocated the same size extent as the input file. Take care - this means that unless there is some spare space in the input file extent there will be insufficient room to create the additional index required by the new file and the conversion process will fail. Normally it is best to code a SIZE statement making an explicit or maximum (999999999) request, ensuring that plenty of spare space is available for the index and the overflow area.

You must supply the length of the key in the indexed sequential file definition KEY LENGTH statement.

15.1.4 Creating an Empty Indexed Sequential File

An empty indexed sequential file is created by calling CONV$ with the input file omitted.

The output file is the new indexed sequential file to be created. In its file definition you must specify that the file organisation is indexed sequential, and you must provide ASSIGN and RECORD LENGTH statements. Unless you supply a SIZE statement and establish a non-zero size other than 999999999 the file will be given the maximum available extent. If you do not supply the key length in the FD then the operator will be prompted for the key length.

Note that if more that a few records are added to an empty indexed sequential file, access will become very slow until the file is reorganised. (See 3.1.2).

15.1.5 Extracting a Relative Sequential File

The conversion routine can be used to extract a relative sequential file from an indexed sequential file. The relative sequential file will have its records in ascending record key sequence. It will contain no logically deleted records.

The input file is the indexed sequential file. Its definition requires only an FD and an ASSIGN statement.

The output file is the new relative sequential file to be produced by the conversion process. In its file definition you must specify that the file organisation is relative sequential and you must provide an ASSIGN statement. Unless you supply a SIZE statement and establish a non-zero size the file will be allocated the same size extent as the input file. Other file definition statements are unnecessary, since the record length is taken from the input file.

15.1.6 Creating a Reorganised Indexed Sequential File

The conversion routine can be used to create a reorganised indexed sequential file from an existing indexed sequential file. The reorganised file will contain no logically deleted records and no overflow chains, and its overflow area will be empty.

The input file is the existing indexed sequential file. Its file definition requires only an FD and an ASSIGN statement.

The output file is the reorganised indexed sequential file to be produced by the conversion process. In its file definition you must specify that the file organisation is indexed sequential and you must provide an ASSIGN statement. Unless you supply a SIZE statement and establish a non-zero size the file will be allocated the same size extent as the input file. Other file definition statements are unnecessary since the record length and record key length are taken from the input file.

Note that if you omit the SIZE statement (or specify a zero size) so as to allocate the output file the same extent size as the input file, then there must be spare space in the existing extent. If this is not the case the conversion process will most likely fail. This is because if there are any overflow records at all the new file is liable to be larger than the old one, since the index area will have grown due to there being more records requiring reference in the prime area. This means that you cannot normally reorganise an indexed sequential file which has been truncated (by CLOSE...TRUNCATE) by using an extent of the same SIZE as the one it currently occupies.

15.1.7 Creating a Relative Sequential File Free of Deletions

The conversion routine can be used to create a relative sequential file containing no logically deleted records from an existing relative sequential file. (Note however that to simply copy one relative sequential file to another it is faster to use the COPY$ system routine described in 15.2.)

The input file is the existing relative sequential file. Its file definition requires only an FD and an ASSIGN statement.

The output file is the new relative sequential file to be produced by the conversion process. In its file definition you must specify that the file organisation is relative sequential and you must provide an ASSIGN statement. Unless you supply a SIZE statement and establish a non-zero size the file will be allocated the same size extent as the input file. Other file definition statements are unnecessary, since the record length is taken from the input file.

15.1.8 Programming Notes

If a relative sequential file is used as either the input or output file for CONV$ then a BLOCK CONTAINS statement in the FD will normally improve the performance of the conversion.

15.2 The File Copy Routine, COPY$

The COPY$ system routine copies an input file to an output file without performing any conversion process. It employs the unused part of the user area following the program last loaded as a buffer area so that it can transfer very large blocks between the files. The routine therefore executes much faster than the conversion routine which has to process a record at a time. On the other hand COPY$ cannot be used to change indexed sequential to relative sequential or drop logically deleted records. However, by copying an indexed sequential file to a larger area its overflow area will be automatically extended without the overhead of reorganising the file.

COPY$ can be used on direct access files but should not be employed to transfer data to a printer since this will not result in a properly formatted report appearing. The routine is not concerned with record format and can be used to transfer any sort of information.

If the size specified is 999999999 the output file extent will be allocated the maximum amount of contiguous space available.

15.2.1 Invocation

The copy routine is invoked by a CALL of the form:

```
CALL COPY$ USING filename-i filename-o
```

Here filename-i is the name of the file definition for the input file and filename-o is the name of the file definition for the output file.

The filename that is associated with filename-i and filename-o must both be closed when COPY$ is called and they will remain closed when the routine returns control. If volume identification checking is specified by the VOLUME phrase appearing in either or both FDs, it will be applied in the normal way when the copy routine opens the file for processing.

If a file with the same name as the output file already exists on the output volume, it will be deleted. The output file is always allocated anew. It is allocated the extent specified in its file definition's SIZE statement except that if the SIZE statement is omitted, or the actual size is specified as zero, the input file extent size will be used.

The file definition for the input file need contain only the FD and ASSIGN statements.

The file definition for the output file must contain the FD and ASSIGN statements and have the same ORGANISATION phrase as the input file. You may, if you wish, code ORGANISATION UNDEFINED in both cases and possibly avoid an unnecessary access method routine being included in your program. Unless you supply a SIZE statement and establish a non-zero size the file will be allocated the same size extent as the input file. Other file definition statements are unnecessary, since any additional file attributes are taken from the input file.

15.2.2 Exceptions

Exception condition 1 is signalled if an irrecoverable I/O error arises on either the input or output file.

Exception condition 2 may be generated for a variety of reasons, identified by the value of $$RES:

- Input file not available ($$RES="3");
- Output file capacity less than that of the input file ($$RES="5").

When control is returned following an exception both FD's will be closed in the normal way. An incomplete output file may be in existence in this case. If so it can be deleted either by using the file utility command program interactively or by using the copy utility again after taking corrective action.

15.2.3 Programming Notes

The size of the output file extent must be at least as great as the extent occupied by the input file. If you wish to reduce the size of the extent occupied by a direct access file you cannot use COPY$. Instead you must use an OPEN OLD followed by a CLOSE TRUNCATE.

When the output file size is greater than that of the input file the extra space is initialised to binary zeros. This automatically extends the overflow area associated with an indexed sequential file.

COPY$ uses the free storage at the end of the user area as a work space. If you invoke the routine from an overlay or chained program you may need to use the FREE$ system routine before passing control to the program to ensure that the System Manager allocates the maximum amount of free space actually available.

15.3 The Catalogue Routine, CATA$

Instead of coding the unit-id and, possibly, the volume-id of a file in its file definition you can hold this information in a special relative sequential direct access file known as the catalogue. Each record of the catalogue contains the file-id, and supplies the unit-id and volume-id, of a catalogued file.

The catalogue maintenance command program ($CATAL) is used to create and update a catalogue, which can contain information for up to 110 files. At run-time the CATA$ system routine is used to extract information from a catalogue and place it in the unit-id and volume-id fields of a list of up to 16 file definitions.

15.3.1 Invocation

The catalogue routine is invoked by a CALL of the form:

```
CALL CATA$ USING catalogue-name filename number
```

Here catalogue-name labels the file definition of the catalogue itself and filename identifies the first file definition of a file list. Number is an integer literal or PIC 9(4) COMP variable whose value specifies the number of file definitions in the file list.

The file definition for the catalogue file need only contain the FD and ASSIGN statements. Its ORGANISATION must be specified as RELATIVE-SEQUENTIAL. Volume identification checking is optional. If specified it will be applied when the catalogue is opened by the routine.

The file list consists of between 1 and 16 file definitions defined in contiguous storage. Data and map definitions may not appear within the list, which can therefore only be constructed using the following file definition statements:

```
FD          KEY LENGTH
ASSIGN      OPTION
BLOCK CONTAINS RECORD LENGTH
KEY         SIZE
```

The catalogue file definition and all files within the list must be closed when CATA$ is called. They will remain closed when the routine returns control.

15.3.2 Processing

The catalogue routine examines the file-id of each file definition in the list. If the file-id is present in the catalogue it completes the unit-id and volume-id fields from the information in the catalogue. Otherwise, if there is no information for the file in the catalogue, the file definition remains undisturbed unless the unit-id has been specified as ? (a question mark followed by two blanks).

Whenever the unit-id of an FD in the list is supplied as ? then CATA$ will prompt the operator for the true unit-id and volume-id if this information is not present in the catalogue. Assigning every catalogued FD to unit ? can therefore simplify system development during the stages when the catalogue is still evolving.

The example shows a typical catalogued FD and the prompt which is output when there is no information present for it:

```
FD CUSFIL ORGANISATION INDEXED-SEQUENTIAL
ASSIGN TO UNIT "?" FILE "CUSTOMER"

$75 SPECIFY CUSTOMER FILE INFORMATION
$75 UNIT-ID:100 VOLUME-ID:SALESV
```

The operator replies, which are underlined, cause the unit-id field in the CUSFIL file definition to be set to "110", and the volume-id field to be set to "SALESV".

15.3.3 Exceptions

Exception condition 1 is signalled if an irrecoverable I/O error is encountered when reading the catalogue.

Exception condition 2 is generated if the catalogue file is not present ($$RES="3"), or if the file specified does exist, but does not have the attributes of a catalogue ($$RES="1").

15.4 The Delete Routine, DELE$

The DELE$ system routine is used to delete a file given its name, which is supplied in an FD.

15.4.1 Invocation

The routine is invoked by a CALL of the form:

```
CALL DELE$ USING filename
```

Here filename is the name of the file definition for the file which is to be deleted. Only the FD and ASSIGN statements are necessary in this definition, which must be closed when the routine is called and will remain closed when it returns control. Any valid organisation may be specified in the file definition and you may, if you wish, code ORGANISATION UNDEFINED to possibly prevent an unnecessary access method routine from being included in your program.

15.4.2 Processing

The routine deletes the file specified by the file-id of the ASSIGN statement, irrespective of whether its file organisation is the same as that specified by the ORGANISATION clause. The file definition is not altered in any way by DELE$.

15.4.3 Exceptions

Exception condition 1 will be signalled if an irrecoverable I/O error arises whilst attempting to delete the file.

Exception condition 2 occurs if the file to be deleted cannot be found on its volume.

15.4.4 Programming Notes

The delete routine is usually employed to remove an unwanted file detected when an OPEN NEW operation on a relative sequential file is suppressed with a file operation exception. It should be used rather than the sequence OPEN OLD, CLOSE DELETE, which has the following disadvantages:

- OPEN OLD will only open files with the same organisation as that specified in the file definition. Therefore the sequence could not be used to delete an unwanted indexed sequential file given a relative sequential FD;

- OPEN OLD will return information from the unwanted file label in the FD and therefore overwrite the size and record length information which may have been established for OPEN NEW.

15.5 The Rename Routine, RENA$

The RENA$ system routine is used to alter the file-id of a direct access file which already exists. The file, which may be of any organisation, is not modified in any other way. If you attempt to rename a print file your program will be terminated in error.

15.5.1 Invocation

The routine is invoked by a CALL of the form:

```
CALL RENA$ USING filename new-file-id
```

Here filename is the name of the file definition for the file which is to be renamed. Only the FD and ASSIGN statements are necessary in this definition, which must be closed when the routine is called and will remain closed when it returns control. The file-id specified in the ASSIGN statement's FILE clause is the old-file-id, and should identify an existing file.

Any valid organisation may be specified in the file definition and you may, if you wish, code ORGANISATION UNDEFINED to possibly prevent an unnecessary access method routine from being included in your program.

The second parameter, new-file-id, is a PIC X(8) variable or 8 character literal which will be used to identify the file once the rename is successful.

15.5.2 Processing

Providing the rename is successful, no exception occurs and the new-file-id becomes the identifier by which the file is subsequently known. In addition this new-file-id replaces the old-file-id in the FD supplied to RENA$. This allows you to issue an OPEN OLD using this FD if you require to access the file once you have renamed it.

15.5.3 Exceptions

Exception condition 1 will be signalled if an irrecoverable I/O error arises whilst attempting to rename the file.

Exception condition 2 may be generated for the following reasons, each of which is identified by the value of $$RES:

- A file with the old-file-id is not present on the unit specified by the FD ($$RES="3");
- A file with the new-file-id already exists on the unit specified by the FD ($$RES="4").

When control is returned following an exception the FD you passed to the rename routine remains unchanged.

15.6 The File Information Routine, FILE$

The file information routine is used to obtain the file-id and unit-id fields of an FD from the operator, who is prompted for this information. The routine is employed extensively by Global Cobol system software and you may wish to use it to standardise the way in which you obtain file information in those programs, such as utilities, which work with many different files.

15.6.1 Invocation

The file information routine is invoked by a CALL of the form:

```
CALL FILE$ USING filename [prefix]
```

where filename identifies the file definition whose file-id and unit-id are required. This file definition must be closed when FILE$ is

called, and will remain closed when the routine returns control. Any valid organisation may be specified in the file definition and you may, if you wish, code ORGANISATION UNDEFINED to possibly prevent an unnecessary access method routine from being included in your program.

If the second parameter, prefix, is absent (the normal case) the routine prompts the operator for the file-id, to be used in the FD, and the unit-id.

When the second parameter, the prefix, is present it must be a PIC X(2) variable or 2 character literal. In this case, if the operator replies with a non-prefixed file-id (one whose second character is not a full stop) the routine manufactures the file-id to be used in the FD from the prefix concatenated with the first six characters of his input. For example, if the prefix were S., and the operator keyed SALES, the file-id actually used would be S.SALES.

The prefix supplied to FILE$ is ignored if the operator keys a prefixed file-id such as B.SALES. In this case the operator's input is used unchanged.

15.6.2 Processing

Before calling FILE$ you should have displayed an explanatory message such as:

```
INPUT MASTER FILE
```

The routine will then output the file-id prompt:

```
:
```

on the same line and accept the file-id keyed by the operator. It will then output the unit-id prompt:

```
UNIT:
```

also on the same line, and accept a unit-id.

A typical dialogue using the routine might therefore appears as:

```
INPUT MASTER FILE:CUSTOMER UNIT:100
```

where the operator replies are underlined.

The operator may reply <CR> instead of supplying a file-id. In this case the processing depends on the value of the file-id in the FD you supplied to FILE$. If this file-id was ? (a question mark followed by 7 blanks) the routine exits signalling exception condition 2 and the FD remains unchanged. If, however, the file-id contains any other value apart from ? the routine leaves this value, which serves as a default, undisturbed and continues by outputting the unit-id prompt.

The operator may reply <CR> instead of supplying a unit-id. If the unit-id you supplied in the FD contains ? (a question mark followed by two blanks) the <CR> reply will not be accepted and the operator will be re-prompted for a valid unit-id. If, however, the unit-id contains any value apart from ? the routine leaves this value, which serves as a default, undisturbed. It then returns control to the caller without signalling an exception.

In essence, therefore, you supply a ? file-id or unit-id when a sensible default cannot be provided, or when you want the operator to be able to signal a special processing requirement by replying <CR> to the file-id prompt.

15.6.3 Exceptions

Exception condition 2 is generated by FILE$ in the event that the FD passed to FILE$ contained a file-id consisting of a question mark followed by seven blanks and the operator keyed <CR> in response to the file-id prompt. This exception, which is the only one generated by the routine, can be used to allow the operator to signal a special processing requirement. For example, keying <CR> instead of a file-id might mean that the operator requires to leave the current program and obtain the ready prompt.

15.6.4 The FILDF$ Routine

An alternative entry point to FILE$, called FILDF$, permits you to show the default file-id and unit-id to the operator. It is invoked by a CALL of the same form as FILE$:

```
CALL FILDF$ USING filename [prefix]
```

and performs the same processing, except that before the prompt for the file-id and the unit it shows the current default value in parentheses. For example:

```
UNIT(xxx):
```

If the value of the file-id or unit-id is ?, indicating that there is no default, then no default is shown.

15.7 The Volume Identification Routine, VOLID$

The VOLID$ system routine is used to obtain the volume-id of the volume currently mounted on a unit. An exception is signalled if the volume does not support a standard System Manager directory and therefore does not possess a volume-id.

15.7.1 Invocation

You may obtain the volume-id of the volume currently mounted on a unit by coding a CALL statement of the form:

```
CALL VOLID$ USING filename
```

where filename identifies an FD which should have the volume-id specified as a symbol if you wish to examine it following the call. (This is unnecessary if you are simply using VOLID$ to check whether or not the volume supports a standard directory.)

Note that the unit-id must be specified in the file definition, which must be closed when VOLID$ is called, and will remain closed once control is returned. Any valid ORGANISATION may be specified in the FD, including ORGANISATION UNDEFINED.

15.7.2 Processing

When VOLID$ returns control, either an exception will be signalled or the volume-id of the currently mounted volume will have been placed in the field you have specified using the FD. Note that this means that

if the volume-id is returned and the file subsequently opened, then volume-id checking will take place unless you zeroise the volume-id field.

15.7.3 Exception Conditions

Exception condition 1 will be signalled if an irrecoverable I/O error occurs.

Exception condition 2 will be signalled if the volume assigned to the unit does not support a standard the System Manager directory (for example, the unit was a printer).

The volume-id field will not be changed if an exception occurs.

15.8 The Assignment Routine, ASSIG$

The ASSIG$ system routine can be used to establish or modify a unit assignment, or determine the unit address currently assigned to a unit-id. The routine can also delete a specified assignment or, more drastically, purge all but the initial permanent assignments set up by the $CUS Permanent Unit Assignments option. You may need to use the purge facility if an assign request signals the exception which indicates that the assignment tables have become full. This routine can also be used to return the full assignment table.

15.8.1 Invocation

You can assign a unit address to a unit-id by coding a CALL statement of the form:

```
CALL ASSIG$ USING unit-id unit-address
```

where unit-id is a 3-character variable or literal specifying the unit-id to be assigned, and unit-address is a 3-character variable or literal containing the unit address to which it is to be assigned. If the unit address is not an integer in the range 110 to 999 or a LAN address consisting of a letter and two digits and if it does not contain "?bb" (as described below) an exception will be generated. If the unit-id is already assigned the old assignment will be changed. Otherwise a new assignment is made and an exception is signalled if this is not possible due to the assignment tables being full.

You can determine the unit address currently assigned to a unit-id by coding a CALL statement of the form:

```
CALL ASSIG$ USING unit-id unit-address
```

where unit-id is a 3-character variable or literal specifying the unit-id whose assignment you wish to determine, and unit-address is a 3-character variable, initially containing "?bb", where b represents a space, in which the unit-address will be returned, overwriting "?bb". If the unit-id is not assigned an exception will be generated.

You may delete an assignment by coding a CALL statement of the form:

```
CALL ASSIG$ USING unit-id
```

where unit-id is a 3-character variable or literal specifying the unit-id to be deleted. An exception will be generated if the unit-id was not assigned.

You may purge the assignment tables, reducing them to the size initially established by the $CUS Permanent Unit Assignments option by coding a parameterless CALL:

```
CALL ASSIG$
```

You can determine the full assignment table by a call of the form:

```
CALL ASSIG$ USING "? " uat
```

where uat is the full assignment table as follows:

```
01 UAT
 02 UATID OCCURS 30 PIC X(3) * Unit-ids
 02 UATAD OCCURS 30 PIC X(3) * Equivalent addresses
```

If all the entries are not used then the entry following the last entry will be set to LOW-VALUES. Note that the structure of the assignment table is not guaranteed for any future releases of System Manager and may be changed.

15.8.2 Exception Conditions

Exception condition 1 will be signalled if the unit address supplied is not "?bb", a number in the range 110 to 999 or a LAN address consisting of a letter followed by two digits.

Exception condition 2 will be signalled if an attempt is made to make a new assignment when the assignment tables are full.

Exception condition 3 will be signalled if an attempt is made to determine or delete the assignment of a unit-id which is not assigned.

Exception condition 4 will be signalled if the assignment table cannot be found on the system.

15.8.3 Programming Notes

ASSIG$ can be used to find the unit address currently associated with an assigned unit-id, and this in turn can be used to determine the device type, as implied by the first character of the address:

UNIT ADDRESS	DEVICE TYPE
1xx	local discrete direct access device on non-separated systems
2xx	local direct access domain or subunit on non-separated systems
3xx	reserved

4xx	reserved	
5xx	printer	
6xx	direct access domain or subunit on the master computer on a network or separated system	
7xx	reserved	
8xx	remote discrete direct access device	
9xx	remote direct access domain or subunit	
cxx	discrete direct access device on network or separated system - the system or computer designated by the initial letter	
Cxx	direct access domain or subunit on a network or separated system - the system or computer designated by the initial letter	

Table 15.8.3 - Unit Addresses and Device Types

You should note that the purge operation simply reduces the assignment tables to the size they initially occupied as a result of the use of $CUS Permanent Unit Assignments. Providing none of the permanent assignments has been modified or deleted this will restore the assignment tables to their initial state, just as if $E were run. However, if the permanent assignments have been altered in any way, the result of the purge operation is not easy to predict. Generally, of course, permanent assignments will not be modified.

15.8.4 Memory Paged ASSIG$

A memory paged version of ASSIG$ is available only when running on System Manager V8.1 and later. This is a version where most of the code for ASSIG$ is held in a System Manager memory page, thus making the subroutine smaller. To link in this routine you must link in subroutine library C.$PAGES as explained in the section on memory paged subroutines in the System Subroutines Manual.

15.9 The Directory Routine, OPEN$, OPENS$, LIST$ & CLOSE$

The directory routine is a system routine with three entry points, OPEN$, OPENS$, LIST$ and CLOSE$. You can use it to determine the file-

id and type of each file present on a System Manager direct access volume, providing it is not in use as a spool unit.

15.9.1 The Directory File Definition

The OPEN$, LIST$, and CLOSE$ entry points each require a filename as their first parameter. This name identifies an FD to be used in the directory processing operation. At a minimum you must code the FD statement and the following ASSIGN statement:

```
FD filename ORGANISATION organisation
ASSIGN TO UNIT unit-id FILE file-id [VOLUME volume-id]
```

You can use any convenient organisation (e.g. UNDEFINED, RELATIVE-SEQUENTIAL) since the one specified is immaterial as far as the directory routine is concerned. You should specify the file-id as a symbol, since LIST$ returns each file-id found to be present in this field. The volume-id should be specified as a symbol if you wish to examine it following a call of OPEN$.

The FD must be closed when it is passed to OPEN$. It will then be opened so that it can be processed by LIST$ or CLOSE$: the type of open is special, however, and prevents the FD from being used by any other file processing operation such as a READ or WRITE. When you have finished examining the directory you must close the FD using CLOSE$. It is then returned to the normal closed state and can, should you so wish, be processed by a conventional access method OPEN statement.

15.9.2 The OPEN$ function

To open the file definition for directory processing you invoke the OPEN$ function with a CALL of the form:

```
CALL OPEN$ USING filename
```

Exception condition 1 will be signalled if an irrecoverable I/O error occurs. Exception condition 2 is generated if the unit you have attempted to open does not contain a volume with a System Manager directory ($$RES = 6), or if it is in use as a spool unit ($$RES = 5). If either exception occurs the OPEN$ does not take effect and the file definition remains unchanged.

If no exception is signalled the volume-id of the currently mounted volume will have been placed in the field you have specified using the FD. Note that this means that if the volume-id is returned and the file subsequently opened normally, then volume-id checking will take place unless you zeroise the volume-id field.

If you attempt an OPEN$ operation on an FD which is already open, either due to an access method OPEN statement or to a previous invocation of OPEN$, your job will be terminated with a stop code.

15.9.3 The OPENS$ Function

The OPENS$ function behaves in the same way as the OPEN$ function except that it will function on a spool unit (that is it will not return exception condition 2 ($$RES = 6)).

15.9.4 The LIST$ Function

To obtain details of the first or next file present in the directory you invoke the LIST$ function with a CALL of the form:

```
CALL LIST$ USING filename type
```

where type is the name of a PIC S9(2) COMP field in which a value defining the file type, as shown in Table 15.5.3, is returned when the routine signals normal completion or exception 3 or 4. In this case the file-id is returned as well, in the field that you named in the FILE clause of the FD's ASSIGN statement.

TYPE	DESCRIPTION
-1 to -99	Backup file. If the type is -1 this is the first file of the cycle, for -2 the second file, and so on.
0	Relative sequential file.
1	Indexed sequential file.
2	Program file or library.
3	Text file.
4	Variable length record file, eg AutoClerk control file.
5	Data library.
6	Compilation file or library (if file has C. prefix), or DMAM database file (otherwise).
7	Global Planner plan file.
8	Global Writer document file.
9	Global Finder database.
11 to 99	User-dependent organisation. Files of type 11 to 99 can be created using the basic direct access method with the appropriate type specified in the ORGANISATION statement used to produce the access method.
110 to 127	System Manager system files. User programs should not attempt to process these files.

Table 15.9.3 - Type Values Returned by LIST$

Exception condition 1 will be signalled if an irrecoverable I/O error occurs, and exception condition 2 when there are no more files present in the directory. In the case of either exception the file-id and type fields are not updated. You should follow either of these exceptions

by issuing a call on the CLOSE$ function to re-establish the FD's initial status and release the System Manager resources involved in the directory processing operation.

Each call on the LIST$ function which completes normally or with exception 3 or 4 returns information concerning a single file from the directory. If the file is not in use (i.e. not already open by the current job or a competing job in a multi-user environment) the routine signals normal completion. Exception 3 is signalled if the file is open as shared, and exception 4 if the file is open exclusively. The file information is supplied in the same order in which it would appear were the directory to be listed using the file utility's LIS instruction: file-ids do not appear in alphabetical order, nor does the sequence of presentation necessarily reflect the time at which a file was created or its position on the volume. Once exception condition 2 is signalled, all file information has been returned to you. If you require to scan the directory again you should follow the CLOSE$ call with a new OPEN$ call and a new sequence of LIST$ calls.

Note that if you attempt a LIST$ operation using an FD which has not been opened by a previous OPEN$ your job will be terminated with a stop code.

15.9.4 The CLOSE$ function

To close the file definition once you have finished processing the directory you invoke the CLOSE$ function with a CALL of the form:

```
CALL CLOSE$ USING filename
```

This simply restores the FD to its unopened state. No actual input/output takes place, so CLOSE$ always completes normally and never signals an exception.

If you attempt a CLOSE$ operation on an FD which has not been opened by a previous OPEN$ your job will be terminated with a stop code.

15.9.5 Programming Notes

Directory operations are relatively slow, so whenever possible you should use conventional access method open operations to determine whether or not files are present. For example, to check whether a file of known type is present it is usually best to issue an OPEN OLD or OPEN SHARED for it.

The directory operations are best employed in applications which are not performance critical, such as displaying or listing file information in response to an operator enquiry, or in printing or conversion operations where a number of files on the same volume are subjected to lengthy processing. In both these cases the time spent accessing the directory is short in comparison with the display or file processing time.

Note that if a spool unit has been opened by the OPENS$ routine the file names returned by the LIST$ routine will be as those listed by $F (i.e. sssoooop where sss is the sequence number, oooo is the operator-id and p is the partition number.

15.10 The File Status Routine, FSTAT$

The FSTAT$ system routine enables you to discover the current size of a file.

15.10.1 Invocation

To obtain the file status you execute a CALL statement of the form:

```
CALL FSTAT$ USING filename size
```

The filename is the name of the file definition, which must be open when the routine is called, and will remain open when it returns control.

The second parameter, size, is the name of a PIC 9(9) COMP field in which the routine returns the size of the file in bytes. (See the programming notes which follow.)

15.10.2 Programming Notes

The size of an RS, TF or VL file is defined as the number of bytes in the used part of the file, delimited by the write next pointer. The size of an IS file includes the prime data area, the index area, and all the overflow records currently developed, but does not take account of any spare space reserved for new records. The size of a DMAM database includes all the extents allocated for use by records or index blocks, but does not take account of any spare space available for further allocation. The size of a BD file is its extent size. These definitions imply that WRITE NEXT statements may increase the size of RS, TF and VL files, and WRITE statements the size of IS and DMAM files, providing free space is available at the end of the extent. BD files are unique in that they can never increase in size following initial allocation.

If you are using FSTAT$ to examine the status of a shared file, you can only rely upon the information if you specify a locking strategy required for any file extending statements, and obtain the appropriate lock yourself before calling FSTAT$. You should obviously unlock the file when the status is no longer important to you.

If you do not (or cannot) do this, then it may be extended by another user after you have obtained the size. This should not be a problem if you are only using the size for a reporting function, or to make a decision about file re-allocation.

15.11 The Set Password Routine, SET$

The SET$ system routine is used to establish a password, so that files which have been secured with that password can be opened. The routine can also be used in conjunction with the SECUR$ routine to secure files by assigning them a password.

15.11.1 Invocation

To set the password you execute a CALL statement of the form:

```
CALL SET$ USING password
```

where password is an 8 character variable or literal containing the password to be set. It should consist of printable ASCII characters, as otherwise it will not be possible to key it into the file utility or in response to the System Manager password prompt. If the password

is all spaces then the current password, if any, will be cleared and there will be no password established.

15.11.2 Programming Notes

The password specified becomes the current password used for checking secured files. It remains in force until a new password is set up, either by another invocation of the SET$ routine or by the operator keying another password in response to the System Manager prompt:

```
PLEASE KEY PASSWORD FOR file-id:
```

which will appear whenever you attempt to open a file which has been secured with a different password. If the password is not changed in either of these ways it will remain current until the end of the session when the $E command, used to sign-off, clears it. The password is only set for the partition in which SET$ is called, and each partition running on a single screen may have its own distinct password set up.

15.12 The Secure File Routine, SECUR$

The SECUR$ system routine is used to secure a file by assigning it the current password, which can be established using the SET$ system routine. SECUR$ can be used to secure an unprotected file, or to change the password on a file, or to remove a password from a file.

15.12.1 Invocation

To secure a file by assigning it the current password you execute a CALL statement of the form:

```
CALL SECUR$ USING filename
```

where filename is the name of a file definition, which must have been opened using OPEN OLD or OPEN NEW before the routine is called, and which will remain open when it returns control.

15.12.2 Programming Notes

The new password is only established when the file is closed. If you fail to close the file the old password will remain in force.

The password on a secured file can be changed as follows:

- establish the old password using SET$;
- OPEN OLD file;
- establish the new password using SET$;
- secure the file with the new password using SECUR$;
- CLOSE file.

If the new password is specified as all spaces then the resulting file will no longer be password secured.

If the file is not open when SECUR$ is called, or if it is shared, then your program will be terminated in error.

15.13 The IS File Size Calculation Routine, CALC$

The CALC$ system routine is used to calculate the size of an IS file required to contain a specified number of records.

15.13.1 Invocation

The routine is invoked by a CALL of the form:

```
CALL CALC$ USING filename records
```

where the filename parameter identifies a closed IS file definition. The record length and key length must have been established, and usually a symbol is defined for the size field, in which the routine returns its result. For example, a typical FD might be:

```
FD MAST ORGANISATION INDEXED-SEQUENTIAL
ASSIGN TO UNIT "DSK" FILE "SAMAST"
KEY LENGTH IS 8
RECORD LENGTH IS 200
SIZE IS Z-SIZE
```

The records parameter is the name of a PIC 9(9) COMP variable in which the number of records the file is to contain has been established.

15.13.2 Processing

The routine calculates the file size required and returns this value in the size field of the FD.

15.13.3 Exception Condition

Exception condition 1 will be signalled if the number of records calculated is too large to fit into a PIC 9(9) COMP variable.

15.13.4 Programming Notes

The calculation will always give a size sufficiently large for the number of records specified, irrespective of whether or not the file contains overflow records. It will, therefore, be a slight over-estimate in most cases.

15.14 Fix Product Serial Number and Expiry Date, FIX$

The FIX$ system routine enables you to fix a serial number and, optionally, an expiry date on a template file so that it becomes a product file. A template file is any program, compilation file or library which has not yet been made a product file. FIX$ lets you perform under program control the functions provided by the file utility's SER and FIF instructions, which are described in the Global System Manager Utilities Manual.

15.14.1 Invocation

To fix a product serial number and, optionally, an expiry date on a template file you execute a CALL statement of the form:

```
    CALL FIX$ USING filename serial [date]
```

The filename is the name of the file definition identifying the template file, which must be closed at the time FIX$ is called, and which will remain closed when the routine returns control. At a

minimum you must code the FD statement and the following ASSIGN statement:

```
FD filename ORGANISATION organisation
ASSIGN TO UNIT unit-id FILE file-id [VOLUME volume-id]
```

You can use any convenient file organisation (e.g. UNDEFINED, RELATIVE-SEQUENTIAL) since the one specified is immaterial as far as FIX$ is concerned.

The second parameter, serial, must be the name of a PIC 9(6) COMP field containing a product serial number in the range 0 to 8,379,999 inclusive. If you supply the value 0 the System Manager serial number of the current system will be used.

The optional third parameter date must, if present, be the name of a PIC 9(6) COMP field containing the desired expiry date in internal format. If the parameter is omitted no expiry date will be assigned.

15.14.2 Exception Conditions

Exception condition 1 will be signalled if an irrecoverable I/O error occurs whilst FIX$ is accessing the template file.

Exception condition 2 is generated if the specified file is not present on the indicated unit (in which case $$RES is "3"), or if a file with the correct name is found, but it is not a template file (when $$RES will be "1").

15.15 The Scratch Volume Routine, SCR$

The SCR$ system routine enables you to scratch the directory of a direct access volume (e.g. a diskette) or a subvolume of a hard disk domain. It results in the deletion of every file occupying the volume or subvolume. The routine cannot be used to scratch an entire domain.

15.15.1 Invocation

To scratch a volume you code a CALL statement of the form:

```
CALL SCR$ USING filename
```

where filename identifies a file definition in which the unit-id of the volume to be scratched has been established. Any valid organisation may be specified in this FD, including ORGANISATION UNDEFINED. The FD must be closed when SCR$ is called, and will remain closed when the routine returns control.

No volume-id checking is performed by SCR$, but if a volume-id is established in the file definition this will be used to rename the volume once it has been scratched.

15.15.2 Exception Conditions

Exception condition 1 will be signalled if an irrecoverable I/O error occurs during the scratch operation.

Exception condition 2 indicates the specified unit was not a direct access device or was a domain.

15.16 Device Information Routine, DEVIN$

The DEVIN$ routine is passed a DN block containing either a unit address or a format code (e.g. A1M, C11A). It signals an exception if the format or unit is not supported. Otherwise, when the routine completes normally, it returns information in the DN block giving the characteristics of the device, and a 25 character device description.

15.16.1 The DN Device Information Block

The device information block is defined as follows:

```
01    DN
 03 DNFORM PIC X(6)  * FORMAT CODE
 03 DNTYPE PIC X     * DEVICE TYPE
 03 DNUAD PIC X(3)   * UNIT ADDRESS
 03 DNVSIZ PIC 9(9) COMP * TOTAL VOLUME SIZE
 03 DNDESC PIC X(25)      * DRIVE DESCRIPTION
 03 DNDLEN PIC 9(4) COMP * LENGTH OF DESCRIPTION
 03 DNNUMU PIC 9(2) COMP * NUMBER OF UNITS
 03 DNDATA PIC 9(9) COMP * DATA SIZE OF UNIT
 03 DNFMTG PIC 9 COMP     * 0=NO FMTG, 1=WHOLE
                       * 2=TRACK,3=TRACK IMAGE
 03 DNCNCD PIC 9(2) COMP * CONTROLLER CODE
 03 DNDRIV PIC 9(2) COMP * DRIVE NUMBER
 03 DNDACC PIC 9(2) COMP * DEFAULT ACCESS OPT
 03 DNSEC PIC 9(4) COMP * SECTOR SIZE
 03 DNSPT PIC 9(4) COMP * SECTORS PER TRACK
 03 DNHPC PIC 9(4) COMP * HEADS PER CYLINDER
 03 DNSTA PIC 9(9) COMP * START OF DIRECTORY
 03 DNDIRS PIC 9(6) COMP * DIRECTORY SIZE
 03 DNDGAP PIC 9(6) COMP * GAP BETWEEN DIRECTORY AND NEXT
                       * TRACK (DISKETTE)
 03 DNVTYP PIC 9(2) COMP * VOLUME TYPE
 03 DNLUAD PIC X(3)  * LAN UNIT ADDRESS
 03 DNNFIL PIC 9(4) COMP * MAXIMUM NO. OF FILES IN DIRECTORY
 03 DNDNGP PIC 9(4) COMP * GAP BETWEEN DIRECTORY AND NEXT
                       * TRACK (DISKETTE)
 03 DNFILL PIC X(2)  * RESERVED FOR FUTURE USE
```

You must set up the DNFORM field to contain either a format code, aunit address, padded to six characters with trailing blanks or an ANAcode (e.g. G1, O2).

15.16.2 Information Returned by DEVIN$

When DEVIN$ signals normal completion the routine returns thefollowing information in the DN block:

DNFORM
If a format code was supplied, then it is returned unchanged in this field. Otherwise, if a unit address was supplied, it is replaced by the corresponding format code. The format can be supplied as an ANA code in which case the first unit with a format of that description will be returned.

DNTYPE The device type code returned is one of:
- T = Tiny capacity diskette (less then 240K)
- L = Low capacity diskette (240 to 480K)
- M = Medium capacity diskette (480 to 960K)
- H = High capacity diskette   (960K or more)
- S = Subvolume of a domain
- D = Domain

```
 B = Big domain (larger than 2GB)
 R = RAM disk
 X = DLV (ie separate volume on a hard disk)
 P = Printer
```

Type B should not be returned on any properly configured domain.

DNUAD If a format code was supplied in DNFORM this will be the lowest unit address corresponding to that format. Otherwise, when a unit address was provided, this field will contain that address. If the format was supplied in DNFORM as an code the first unit with a format of that description will be returned. If this is a Unix system then not only the devices associated with this system are searched for the format, but also the devices attached to system A.

DNVSIZ The total size of the volume corresponding to the unit, including reserved areas, alternates and directory space. The total number of cylinders can be calculated by dividing this number by the sector size, number of sectors per track and number of heads per cylinder.

DNDESC The device description. Note that this description only identifies the location of the drive, and not the precise format being used. For example, single and double density formats on the same device will return an identical device description, such as "LEFT HAND DRIVE". Descriptions shorter than 25 characters are padded with trailing blanks.

DNDLEN The length of the device description, excluding trailing blanks. The length is always at least 1.

DNNUMU For a domain, this is the number of subunits on the domain. For a subunit, it is the index to the subunit within the domain, for example subunit 237 of domain 220 will return an index of 17. For all other types of unit it is the number of consecutive units, starting with DNUAD, which have the same format code. Thus if units 140 and 141 are the same format but 142 is either undefined or a different format, unit 140 willreturn a value of 2 in DNNUMU and 141 a value of 1.

DNDATA The total data capacity of the unit, available for allocation as files (or for allocation as subunits if the unit is a domain). Note however that file allocations are always rounded up to whole number of sectors, and subunits to a whole number of tracks.

DNFMTG Formatting flag, returned as one of:

```
 0 = formatting not supported
 1 = whole volume formatting only
 2 = individual track formatting
 3 or greater = individual track formatting using a
 supplied track image.
```

For a printer this field is set to the device characteristic byte, bit #40 of which is set for X-ON/X-OFF printers.

DNCNCD A code which identifies the System Manager controller used to access the device except when the device is a printer.

DNDRIV The physical drive number of the device (not necessarily unique if several different controllers are used).

DNDACC The default access option used by the System Manager, chosen to give optimum performance when reading or writing large records.

DNSECT The sector size in bytes for a disk, or the width in characters of a printer.

DNSPT The number of sectors on each track for a disk, or the page size in lines of a printer (0 = hardware form feed used).

DNHPC The number of heads on each cylinder, or the baud rate for a printer.

DNSTA The byte address of the start of the directory area within the volume.

DNDIRS The size of the directory in bytes (a whole number of sectors).

DNDGAP For a diskette, the size in bytes of gap between the end of the directory and the start of the next track. This field is undefined for domains, subvolumes and printers. THIS FIELD FOR INTERNAL USE BY GLOBAL BUSINESS SYSTEMS.

DNVTYP The volume type code, a number used to differentiate between different format diskettes readable on a single drive.

DNLUAD The LAN format of the unit address. Set to spaces if the unit is not on a network or separated system file server, otherwise set to the LAN form of the address (starting with A - Z or a - z) which can be used to access the unit from any computer.

DNNFIL If the device type is a diskette of subvolume, then the maximum number of files that can be allocated on the volume is returned in DNNFIL. For subvolumes this value is normally either 99 or 250.

DNDNGP For a domain, the size in bytes of gap between the end of the directory and the start of the next track. This value must be subtracted from DNDATA to obtain the usable data size. This field is undefined for subvolumes, diskettes and printers.

Note that for printers the only fields that are established by DEVIN$ are DNTYPE, DNDESC, DNDLEN, DNSECT, DNFMTG, DNCNCD and DNSPT.

15.16.3 Invocation

After setting up the DNFORM field to contain either a format or a unit address you invoke the DEVIN$ routine with a CALL statement of the form:

```
CALL DEVIN$ USING DN
```

Providing the system on which it is executing supports the format or unit, the routine completes normally and returns device information in the other fields of the DN block, as explained above. However, an exception is signalled if the format or unit is not supported, or if the value supplied in DNFORM is invalid.

15.16.4 Programming Notes

On a System Manager V5.0 system the controller code and full device description are not available. Therefore, on such systems the

controller code is always set to zero, and a device description of the form "DRIVE NUMBER nn" or "PRINTER" is returned. For a System Manager V5.1 and greater the controller code is only zero for printers.

Some diskette formats contain a few tracks which have a different density or sector size to the rest of the diskette and hence cannot be used for the System Manager directory or for data. For the purpose of the fields returned by DEVIN$, such as the volume size, these tracks are treated as if they were formatted in the same way as the rest of the volume.

Two different unit addresses correspond to the same physical drive only if they have the same controller code, drive number and device description.

If a diskette unit allows track image formatting (DNFMTG > 2) then it supports an extended format as used by the System Manager V5.1 product distribution system.

Since the DEVIN$ routine never accesses the volume mounted on the drive, it cannot detect unallocated subvolumes, and the values returned in DNDATA and DNDIRS refer to the last subvolumes accessed on that subunit. These values are meaningless if no volume has been accessed on that unit since the computer was bootstrapped. If you want to determine whether a subvolume is allocated, and its data size, you should first access the subvolume by calling the VOLID$ routine. Set up an error intercept in the FD you pass to VOLID$ in order to suppress error messages (the intercept should be an EXIT WITH 1 statement). If an I/O error is signalled, and $$RES is set to C, then the subvolume is not allocated. Any other errors represent genuine I/O errors accessing the volume. If the operation completes normally, you can call DEVIN$ to determine the data size.

For Unix systems DEVIN$ not only searches the current system for the unit information but also searches devices attached to system A.

15.17 ISAM Records In Use Routine, ISUSE$

ISUSE$, the ISAM records in use routine, returns the number of records in use for a specified ISAM file and, optionally, the number in the overflow area. Note that both counts include deleted records that have not yet been removed using CONV$.

15.17.1 Invocation

ISUSE$ is invoked by a call of the form:

```
CALL ISUSE$ USING fd area used [overflow]
```

where fd is the file definition of an open IS file which will be left open on return, area is a work area large enough to contain a record of the file, used is a PIC 9(9) COMP variable in which the number of records used is returned, and the optional overflow is a PIC 9(4) COMP variable in which the number of records in the overflow area is returned.

15.17.2 Exceptions

Exception condition 1 will be signalled if an irrecoverable I/O error occurs.

15.18 The Shared Lock Routine, SLOCK$/SULOC$

The Shared Lock routine allows you to obtain a lock on a record similar to that granted by the LOCK verb, but which may also be obtained by other users who also use a shared lock. This would typically be of use in a master/servant record structure, where you wish to find the master record before creating or updating the servant, and lock it to prevent deletion by some exclusive process without denying other users the ability to create different servants from the same master record.

15.18.1 Invocation

This routine is invoked by a call of the form:

```
CALL SLOCK$ USING fd region
```

to gain a shared lock, and released by a call of the form:

```
CALL SULOC$ USING fd region
```

where the fd is the FD on which the lock is to be performed (which must be open), and region is the region to be locked.

15.18.2 Exceptions

Exception 2 will be signalled if an attempt is made to gain a shared lock which someone else already has exclusively.

This routine is fully compatible with (and will work on) all versions of the System Manager.

15.18.3 Programming Notes

The most common situation where shared locks are useful is when processing records which have a master/servant relationship. For example, if the requirement for a system is that each customer record have a pre-existing territory record corresponding to the territory of that customer, then the territory record is a master for the customer record. When adding a customer record you might wish to LOCK the territory record, so as to ensure that it is not deleted. However, if you simply issue a normal LOCK, then not only would it be impossible to delete the record, but it would also be impossible to add another customer record on that territory. The solution would be to issue a shared lock, using SULOC$, permitting other record addition processes to proceed.

Note that if a program which has obtained a shared lock fails to release it for some reason (due to program failure for example), then the shared lock will remain in existence until all users of the file have closed it.

15.18.4 Memory Paged Routine

A memory paged version of the shared lock routine is available only when running your program on V8.1 or later System Manager. This is a version where most of the code for the shared locking routine is held in a System Manager memory page. It can be linked into your program by linking compilation library C.$PAGES as explained in the section on memory paged routines in the system subroutines manual.

15.19 Work Unit Locking, LWORK$ and UWORK$

The work unit locking routine enables you to lock a work unit rather than a file, typically so that your program can get exclusive access to the unit.

15.19.1 Invocation

This routine is invoked by a call of the form:

```
CALL LWORK$ USING fd
```

where fd is the FD of the work unit.

To unlock the unit the following call is used:

```
CALL UWORK$ USING fd
```

15.19.2 Exceptions

Exception 1 is returned if the unit table is full or, when invoking UWORK$, if the unit is not locked. Exception 2 is returned if someone already has the unit locked.

15.19.3 Programming Notes

The lock does not prevent other users from accessing the unit, so it is vital that when unit locking is used all the programs lock the unit before use. The routine is used by Global Business Software modules to control access to the work units they use.

It is important to note that separate FD's are required for every unit locked using LWORK$. The same FD must not be used to lock two different units.

15.20 The Open File with Optional Delete Routine, OPDE$

The OPDE$ routine permits you to open a new file, prompting the operator for deletion if a file with the same name already exists on the unit in question. Additionally the routine detects that it is being run under Job Management, and in this case automatically takes the decision to delete the offending file.

15.20.1 Invocation

This routine is invoked by a call of the form:

```
CALL OPDE$ USING filename [newline]
```

where filename identifies the file definition which is to be opened, and newline is an optional PIC X(4) parameter, used as described under processing below.

15.20.2 Processing

OPDE$ first attempts an OPEN NEW on the file definition provided, using whatever access method is specified by its ORGANISATION statement. If this should be a file organisation which does not support an OPEN NEW operation, such as ISAM or DMAM, then your program will be terminated in error.

If the OPEN succeeds, then OPDE$ immediately returns control to the calling program. However, if the OPEN fails, because there is already

a file with the same name on the unit specified in the ASSIGN clause in the FD, then the operator will be prompted:

```
FILE ALREADY EXISTS - DELETE?:
```

If no newline parameter was supplied, then this prompt is displayed on the same line as the cursor currently occupies (as if a DISPLAY ... SAMELINE had been executed). If a newline parameter was supplied, then it is displayed first, on a new line. When working in formatted mode it is usual to provide a parameter of four spaces, so that the prompt will appear on the baseline.

If the operator replies Y to the prompt, then the file will be deleted, and OPDE$ will proceed to open the new file required. If the operator replies <CR>, N, or any reply except Y then OPDE$ will signal an exception to the calling program.

15.20.3 Exception Conditions

Exception 1 is returned if an irrecoverable I/O error arose when attempting to open the new file.

Exception 2 is returned if the operator declined to delete an existing file.

15.20.4 Programming Notes

OPDE$ is used by several System Manager system programs which wish to be run under Job Management, and issue prompts offering deletion of existing listing files etc. It is only really valuable when the task it is called by is likely to be run under Job Management.

15.21 The Copy Library Index Routine, LIBR$

The LIBR$ system routine is used to build an index of the names and starting positions of the books in a Global Cobol copy library. Subsequently individual books can be accessed using the text file access method by performing a READ statement followed by READ NEXT statements.

15.21.1 Invocation

The routine is invoked by a CALL of the form:

```
CALL LIBR$ USING filename LI [FM]
```

The filename is the name of the text library file, which must have text file organisation and must be closed when the routine is called, and will remain closed when the routine returns control.

The parameter LI is the name of a control block in which the index will be built, and consists of: the name table; a filler to hold the terminating zero byte if the library contains the maximum of 110 books; and the start table. The format is:

```
01 LI
 03 LINAME     OCCURS 110 PIC X(2)  * NAME TABLE
 03 FILLER           PIC X     * MAX. TERMINATOR
 03 LISTART    OCCURS 110 PIC 9(6) COMP * START TABLE
```

When LIBR$ returns control it will have placed the names of the books the library contains in the name table in the order in which they were

found. A corresponding entry in the start table will indicate the starting character number of the first line following the book start line. The last entry returned in the name table will be terminated by a binary zero byte so that a Global Cobol SEARCH can be employed to rapidly check for the presence of a book name.

The optional third parameter, FM, can be provided if spare memory is available and it is desired to optimise the performance of LIBR$ by supplying it with a large additional work area which it can use when scanning the text file. When the third parameter is used the quantity FM should label a free memory request block which has been previously set up to address an area acquired by the FREE$ system routine, as explained in 5.2.2. The block is therefore of the form:

```
01    FM
 03 FMFUN PIC 9 COMP * WILL CONTAIN 2
                     * I.E. GET WORK SPACE
 03 FMSIZE PIC 9(6)  * SIZE REQUIRED
 03 FMALL PIC 9(6) COMP * SPACE ACTUALLY
                     * ALLOCATED
 03 FMPTR PIC PTR    * POINTER TO SPACE
```

If an FM area is not supplied, or if the space actually allocated in FMALL is less then 512 bytes, LIBR$ will simply use the 512-byte FD extension as its work area and no performance optimisation will take place.

15.21.2 Processing

The copy library is scanned sequentially. It is checked to be of the correct format, as described in Appendix F of the Global Cobol Language Manual. Each time a book start line is found a count is incremented by one and used to index the name table and the start table. The book name is placed in the name table and the character number of the line following the book start line is placed in the start table.

15.21.3 Exceptions

Exception condition 1 will be signalled if an irrecoverable I/O error arises while scanning the library file.

Exception condition 2 will be generated if the file cannot be found, or does not have text file organisation.

Exception condition 3 will occur if the text file is not in correct copy library format (for example, a book end line is missing) or a 111st book is encountered. A message giving the name of the book in error will be displayed on the screen.

When control is returned following an exception the FD will be closed and the index table will contain a valid index to the part of the file already processed prior to the exception being detected.

15.21.4 Programming Notes

Once the library index has been constructed you read the lines of a particular book as follows:

- Use the Global Cobol SEARCH statement to find the index of the book name within the name table;

- Move the correspondingly indexed start table entry to the key field of the text file FD used to access the library;
- READ the first line of the book (i.e. the line immediately following the book start line);
- Obtain the second and subsequent lines of the book using READ NEXT.

The book is delimited by the book end line. Unlike any other line within the book, the first significant character of the book end line is an ASCII full stop. Thus to check for the book end line the simplest method is to redefine your record area as a table of single-character entries and check the entry whose number is returned in the TXSIG field to see whether it contains a full stop.

Note that a copy library may contain a null book, consisting of just a book start line immediately followed by a book end line. In this case the READ statement used to access the very first line of the book will obtain the book end line, and, in consequence, subsequent READ NEXTs will not be required.

15.22 The Volume description routines, GTDES$/PTDES$

The volume description routines allow you to read or write volume descriptions. These routines would typically be of use in installation programs or as an addition to general volume description information.

15.22.1 Invocation

The reading of a volume description is invoked by a call of the following form:

```
CALL GTDES$ USING fd description
```

To write a volume description you code the following statement:

```
CALL PTDES$ USING fd description
```

where the fd is an FD containing the unit-id of the volume and description is a PIC X(50) field in which the volume description will be returned or in which the new volume description is supplied.

15.22.2

Exception condition 1 will be returned if an I/O error has occurred when accessing the volume description table or if the volume does not have an entry in the volume description table.

15.22.3 Programming Notes

Domains initialised using pre-V7.0 versions of System Manager may not have space in the volume descriptions file for all the possible volume on the domain. This means that some of the higher numbered volumes will not have volume description entries and an error condition 1 will be returned if an attempt is made to access entries for these volumes.

15.23 The Sub-volume size routine, SUBS$

The SUBS$ routine returns the current size and maximum possible size of a subvolume.

15.23.1 Invocation

To return the subvolume sizes you code a CALL statement of the form:

```
CALL SUBS$ USING unit current maximum
```

where unit is the subvolume unit-id, current is a PIC 9(9) COMP field in which the current size of the subvolume is returned, and maximum is a PIC 9(9) COMP field in which the size of the largest free space available on the domain to allocate a new subunit is returned. If the subunit is not allocated the current size is returned as zero.

15.23.2 Exception conditions

An exception condition of 1 is signalled if the unit supplied is not supported.

An exception condition of 2 is returned if SUBS$ is attempted on a domain.

16. The Data Security System Routines

This chapter describes two routines, SAVE$ and REST$, which can be used to take backups of files and to restore them. The Data Security System, of which these routines form part, is described in detail in Chapter 4 of the Utilities Manual.

You should note that SAVE$ and REST$, like the sort, temporarily acquire whatever free memory is available, and operate faster the more they can obtain. At least 2 Kbytes should be available, otherwise performance may prove very poor. The extra storage is released when the routine in question returns control. There is a full discussion of free memory management in Chapter 8 of the System Subroutines Manual.

16.1 Save Files on Backup Cycle Routine, SAVE$ and SAVEN$

The SAVE$ system routine is used to create a backup cycle containing the files defined in the principal master volume's security catalogue. The principal master volume involved, and the particular cycle to be saved, are specified in parameters supplied to the routine. You can also determine the title that is to be given to the backup cycle when it is first produced, and which is checked to be present whenever the cycle is reused. Alternatively, if you supply no title, the operator will be prompted for one when the cycle is initially created, and asked to confirm that the title is as expected whenever the cycle is subsequently used.

The SAVEN$ routine is identical to SAVE$ except that the backup diskettes are not verified. We do not recommend its use, but in some circumstances it will save time at the expense of security.

16.1.1 Invocation

The save routine is invoked by a call of the form:

```
CALL SAVE$ [or SAVEN$] USING filename SR ["SHARED"]
```

The filename parameter labels a file definition for your security catalogue. Only the FD and ASSIGN statements are required. The ORGANISATION should be specified as RELATIVE-SEQUENTIAL. In the ASSIGN statement the file-id must be the name of the catalogue, and the unit-id and volume-id must be those of the principal master volume on which it resides. The FD must be closed when SAVE$ is called and will remain closed when it returns control.

The SR parameter is the name of a save/restore control area of the form:

```
01 SR
02 SRCYCLE     PIC X      * CYCLE-ID
02 SRTITLE     PIC X(30) * TITLE (OPTIONAL)
02 SRVOLN PIC 9(2) COMP * BACKUP VOLUME NO.
02 SRFILE PIC X(8) * FILE-ID
```

Before invoking SAVE$ you must establish the cycle-id of the backup cycle that you wish to create in SRCYCLE. You must place the title to be assigned or checked in SRTITLE. However, if SRTITLE remains blank the operator will be prompted to assign the title when the backup

cycle is first created, and to confirm it whenever the cycle is re-used.

The other fields of the SR area are used to supply information to your program when SAVE$ returns control.

The optional third parameter is of importance in multi-user and networking environments where a number of user programs can be accessing files at the same time. When it is omitted, or is anything other than the character string "SHARED", SAVE$ opens each of the files to be preserved with an OPEN OLD statement, implying that the prompt:

```
* IN USE ERROR ON description unit - RETRY?:
```

will appear if any such file is currently opened by another user. If this prompt appears the operator will either have to abort the save operation or wait until all other programs have finished with the file in question.

By coding "SHARED" as the third SAVE$ parameter you can request the routine to open each file to be preserved with an OPEN SHARED so that other users can continue inspecting it throughout the backup process. Obviously an inconsistent backup will be taken if updates are taking place at this time. Therefore applications which use this option should ensure that no other users can update the files involved in a save operation whilst SAVE$ is working. When only one update program can be active at once a simple technique is to define a regional lock associated with one of the files to mean "updating allowed". Then the program responsible for the backup operation need only acquire this lock before executing the shared save and all will be well. For example:

```
LOCK SAMAST "FILE"
ON EXCEPTION
DISPLAY "AN UPDATE IS IN PROGRESS"
DISPLAY "- SAVE CANNOT TAKE PLACE" SAMELINE
BELL
STOP RUN
END
CALL SAVE$ USING SECURE SR "SHARED"
etc etc
```

Note that whether or not "SHARED" is specified you must ensure that there are no files opened on the backup unit, $B, before calling SAVE$, otherwise "file in use" I/O errors will occur.

16.1.2 Processing

The save routine is used internally by the $SAVE command program, and you will find a detailed description of how the operator may respond to the prompts it issues in the Global System Manager Utilities Manual.

SAVE$ begins by opening the security catalogue to determine the location of the master files, which are processed one by one in catalogue entry order. A mount prompt appears whenever a master volume is required which is not online.

A backup prompt of the form:

```
MOUNT BACKUP backup-id ON description unit AND KEY <CR>:
```

is output to control the loading of each backup volume. This prompt will be suppressed if the operator correctly anticipates the mounting of the first backup volume. If the SRTITLE field supplied to SAVE$ was blank, the operator will be prompted for the title when the first volume of the backup cycle is new:

```
Backup title:
```

or, if the cycle is already in use, he or she will be asked to confirm that it can be overwritten:

```
Overwrite title backup of date time?:
```

When SRTITLE is non-blank these two prompts are suppressed. The title is then automatically inserted when the cycle is created, and checked whenever it is reused.

If an irrecoverable I/O error occurs on a backup volume, then, following the operator's reply of N to the retry prompt, the continuation prompt:

```
CONTINUE WITH NEW BACKUP VOLUME?:
```

appears to allow the operator to replace the faulty volume and continue.

If the save operation succeeds, SAVE$ returns control signalling normal completion. In this case SRVOLN contains the volume number of the last backup volume created, so that you can output this statistic in a report, should you so wish.

16.1.3 Exceptions

If the save operation fails, one of four different exception conditions will be signalled to indicate the type of error that has occurred.

Exception condition 1 can only arise if you are re-using a security cycle. It indicates that the backup title is not as expected. Either it does not match the information you supplied in SRTITLE, or if your SRTITLE was blank, the operator did not reply Y to the overwrite confirmation prompt.

Exception condition 2 indicates that SAVE$ has detected that it is about to request that a master file be mounted on the backup unit itself. This means either that the security catalogue is in error, or that the wrong unit-id was supplied in the file definition passed as a parameter to SAVE$. The field SRFILE contains the file-id of the master file involved.

Exception condition 3 occurs if the operator replies N to any mount prompt, if there is an irrecoverable I/O error on a master file, or if a master file defined in the security catalogue is not present on its master volume. SRFILE contains the file-id of the file involved. If this is the name of your security catalogue, then either there was an irrecoverable I/O error accessing the file, or the file was not

present on the principal master volume, or it was present, but was not a valid catalogue.

Exception condition 4 is signalled if the operator indicates that a required backup volume is not available, either by replying N to the backup prompt or to the continuation prompt which follows a permanent I/O error on a backup volume. The field SRVOLN contains the volume number of the backup volume involved.

16.1.4 Programming Notes

The only console output from SAVE$ consists of the prompts described under "Processing" above. The calling program is responsible for reporting that the save operation has been completed successfully, or producing an appropriate warning if it has been terminated in error. In production running, assuming a debugged system in which the calling program, rather than the operator, supplies the backup title, the four exceptions may be interpreted as follows:

1 This should not occur providing sensible operating procedures have been adopted. If it does it means that two or more different versions of the same cycle of backup volumes are in existence, clearly a situation to be avoided. The save operation must be repeated using the correct version.

2 This should not occur.

3 A permanent I/O error has arisen on a master volume, or a master volume has become lost. If the master data is truly unobtainable it will be necessary to restore an earlier version from a previous backup cycle and repeat the intervening processing before attempting the save operation again.

4 The operator has terminated the save operation because he needs a spare backup volume and one is not available. The save operation can be repeated as soon as the necessary volume or volumes have been initialised.

16.2 Restore Files from Backup Cycle Routine, REST$

The REST$ system routine is used to restore master file information from a previously saved backup cycle. The principal master volume involved, and the particular cycle to be restored, are specified in parameters supplied to the routine. You can also specify a title which must be present on the backup cycle in order that the restore operation be allowed to proceed. Alternatively, you may supply no title and in this case the operator will be prompted to confirm that the correct backup cycle is in use.

16.2.1 Invocation

The restore routine is invoked by a call of the form:

```
CALL REST$ USING filename SR [type]
```

The filename parameter labels a file definition for your security catalogue. Only the FD and ASSIGN statements are required. The ORGANISATION should be specified as UNDEFINED. In the ASSIGN statement the file-id must be the name of the catalogue, and the unit-id and volume-id must be those of the principal master volume on which it

resides. The FD must be closed when REST$ is called and will remain closed when the routine returns control. Note that REST$ uses the filename parameter solely to determine the identity of the principal master volume: the restore process is controlled by file, volume and unit-id information established on the backup volumes themselves during the save process.

The SR parameter is the name of the save/restore control area, the format of which is defined in section 16.1.1. Before invoking REST$ you must establish the cycle-id of the backup cycle you wish to restore in SRCYCLE, and place the title you wish to have checked in SRTITLE. Alternatively you may leave SRTITLE blank, and in this case the operator will be prompted to confirm that the title of the backup cycle is as expected before the restore operation is allowed to proceed.

The other fields of the SR area are used to supply information to your program when REST$ returns control.

The optional third parameter, type, is a PIC X variable which may contain one of the following values:

F to indicate that a full restore is required (this is the default if the parameter is omitted);

L to indicate that the backup volume's contents should be listed on the screen (which must be in teletype mode);

S to indicate that a selective restore is required; this is not generally used except in the context of program development.

Note that before calling REST$ you must ensure that there are no files open on the unit assigned to $B, otherwise the restore process will be terminated with I/O error F, indicating that there are files already in use on the backup device.

16.2.2 Processing

The restore routine is used internally by the $RESTORE command program, and you will find a detailed description of how the operator may respond to the prompts it issues in the Global System Manager Utilities Manual.

REST$ begins by requesting the first backup volume of the cycle by a backup prompt of the form:

```
MOUNT BACKUP backup-id ON description unit AND KEY <CR>:
```

Each subsequent backup volume is requested by a similar prompt.

If the SRTITLE field supplied to the routine was blank, then, as soon as the first backup volume is mounted, the operator is asked to confirm that the actual title and date are as expected:

```
Restore backup title of date time?:
```

When SRTITLE is non-blank this prompt is suppressed and the title you have supplied is simply checked against the one actually present on the backup cycle.

If type is F or has been omitted indicating a full restore, then each backup file is read from the backup cycle and a mount prompt is issued if the required master volume is not online. I/O errors, arising on either master or backup files, are handled by the normal System Manager mechanism, the retry prompt.

If type is L then the backup cycle is read through and the names and destinations of the files found are displayed on the screen but they are not restored. This is useful when you need to know what files are within a backup cycle and which volumes they are on.

If type is S, this indicates that a selective restore is required (usually to cope with a partially corrupted master volume). As with list, the names and destinations of the files contained in the backup cycle are displayed, but the operator is given the option to restore these files (Y is keyed to restore, N or <CR> if not). When the first file is to be restored onto a new master volume the operator is given the option to scratch the master volume.

If the restore operation succeeds REST$ returns control signalling normal completion. The field SRVOLN contains the volume number of the last backup volume accessed.

16.2.3 Exceptions

If the restore operation fails one of four different exception conditions will be signalled to indicate the type of error that has occurred.

Exception condition 1 arises if the backup title is not as expected. Either it does not match the information you supplied in SRTITLE, or if your SRTITLE was blank, the operator did not reply Y to the restore confirmation prompt.

Exception condition 2 occurs if REST$ detects that it is about to request that a master file be mounted on the backup unit itself. This probably means that the wrong unit-id was supplied in the file definition passed as a parameter to REST$. The field SRFILE contains the file-id of the master file involved.

Exception condition 3 occurs if the operator replies N to any mount prompt or if there is an irrecoverable I/O error affecting the master file. SRFILE is set to the file-id of the affected master file.

Exception condition 4 is signalled if the operator indicates that a required backup volume is not available by replying N to the backup prompt, or if an irrecoverable I/O error occurs on a backup volume. Exception condition 4 will also arise if REST$ detects that the backup cycle is internally inconsistent, as might be the case if it was only partially created because the previous save operation terminated in error. The field SRVOLN contains the volume number of the backup volume found to be faulty.

16.2.4 Programming Notes

The only console output from REST$ consists of the prompts described under "Processing" above. The calling program is responsible for reporting that the restore operation has completed successfully, or producing an appropriate warning if it has been terminated in error. In production running, assuming a debugged system in which the calling

program, rather than the operator, supplies the backup title, the four exceptions may be interpreted as follows:

1 This should not occur providing sensible operating procedures have been adopted. If it does it means that two or more different versions of the same cycle of backup volumes are in existence, clearly a situation to be avoided. The restore operation must be repeated using the correct version.

2 This should not occur.

3 A permanent I/O error has arisen on a master volume, or a master volume has become lost. The restore operation must be repeated once a new volume has been initialised, or a missing volume found.

4 The backup data itself is in error. It will be necessary to fall back to an earlier backup cycle and then repeat the restore operation.

17. The Multi-Key Sort

The $SORT command described in the Global Utilities Manual allows you to sort files of fixed length records into an order determined by up to nine keys, each of which may be defined as either character or computational fields of ascending or descending sequence. The Global Cobol SORT, RELEASE and RETURN statements defined below allow you to construct sorts of almost any complexity under program control. Before coding your own sort, however, you should check that the same effect cannot be achieved more easily by using $SORT, possibly tailored for a particular purpose, using Global Cobol Job Management.

The sort employs the unoccupied part of the user area following the last program loaded as a work area. The size of the work area determines how many records can be sorted without requiring a work file and the efficiency of the sort when a work file is present. The larger the work area can be, the better.

If you provide a work file and the work area is large enough to allow all the records to be sorted internally, the file will not be used.

You can use the $CALC command described in the Utilities Manual to calculate sort work space requirements, given the number of records to be sorted. A work file, if it is needed, must be large enough to contain all the records and all the keys to be sorted.

17.1 Invoking the Sort

The sort is invoked by the SORT statement, which supplies the addresses of an input and an output routine. The input routine supplies records to the sort using the RELEASE statement, and the output routine obtains sorted records from the sort using the RETURN statement.

17.1.1 The SORT Statement

To initiate the multi-key sort you must code a SORT statement of the form:

```
SORT sc input output [filename]
```

The parameter sc is the name of a sort control area of the following format:

```
01 SC
 03 SCRLEN      PIC 9(4) COMP * RECORD LENGTH
 03 SCRECS      PIC 9(9) COMP * NO. OF RECORDS
 03 SCKLEN      PIC 9(4) COMP * TOTAL KEY LENGTH
 03 SCKEYS      PIC 9(4) COMP * NO. OF KEYS (n)
 03 SCDESC OCCURS n PIC X(8) * KEY DESCRIPTOR TABLE
```

You must initialise all fields of the area, which is read-only as far as the sort is concerned. The size in bytes of the fixed length records must be in SCRLEN and must not be greater than 2000 bytes.

The field SCRECS may be supplied as zero if the number of records to be sorted is not known. If a positive value is supplied the sort will check whether it has sufficient capacity to process the indicated number of records and will signal an exception if this is not the case.

SCKLEN must be set to the total size of all the keys involved in the sort, in bytes. SCKEYS must contain a value between 1 and 9, indicating the number of different keys, which must correspond to the number of key descriptor entries in the following SCDESC table. Each key descriptor consists of a string of exactly 8 characters, of the form:

```
"kkfsoooo"
```

where:

kk is the length of the key in bytes. For ascending character keys kk must be between 01 and 99 inclusive, and for other keys kk must be between 01 and 08 inclusive.

f indicates the format of the key, X=character, C=computational.

s is the sequence indicator, A=ascending, D=descending.

oooo specifies the origin of the first byte of the key within the record. The first byte of the record is 0001.

Thus the descriptor "08XA0005" defines an 8-byte character key, to be sorted into ascending sequence, starting at byte 5 of the record. The first entry in the key descriptor table should be for the most significant key, the second entry for the next most significant... and so on.

The second parameter of the SORT statement, input, is the name of a section within the calling program which is responsible for providing the records to be sorted using the RELEASE statement.

The third parameter, output, is the name of another section which obtains sorted records using the RETURN statement. The System Manager PERFORMs the input section to obtain the records to be sorted, and then the output section to return the sorted records to your program.

The final, optional parameter, filename, identifies the file definition of the sort work file, which is only needed when there is insufficient main memory available to perform the sort internally. Only the FD and ASSIGN statements are required in the file definition, which should be coded:

```
FD filename ORGANISATION UNDEFINED
ASSIGN TO UNIT unit-id FILE file-id [VOLUME volume-id]
```

This file definition must be closed when the SORT statement is executed, and will remain closed when Global Cobol returns control to the next statement following the SORT statement.

Note that if the work file already exists when the sort is called, it will be overwritten if it is needed but will remain in existence at the end of the processing. If the work file is not allocated at the start of the sort it will be deleted at the end of the sort.

17.1.2 Exception Conditions from SORT

When the sort completes normally control is returned to the statement immediately following the SORT statement and no exception is

signalled. However, if you specified a non-zero number of records in the SCRECS field of the sort control area, the System Manager will check that there is sufficient space to perform the sort and signal an exception if this is not the case, or if an irrecoverable I/O error occurs when examining the sort work file (if there is one). You should trap and process the condition using an ON EXCEPTION statement immediately following the SORT statement.

Three separate exceptions may be signalled:

$$COND=1 means that an irrecoverable I/O error occurred when checking the work file capacity;

$$COND=2 indicates that either there is no work file, and there is not sufficient internal memory available to perform the sort, or there is a work file, but it is not sufficiently large;

$$COND=3 indicates that the capacity of the sort algorithm has been exceeded. There is insufficient internal memory available. You must make more user area space available by one of the techniques suggested in chapter 8 of the Global Cobol System Subroutines Manual.

An exception will also be signalled if the input or output routine executes an EXIT WITH condition statement. The condition number signalled by the SORT will be that in the EXIT WITH statement. To avoid confusion with those exceptions generated by the SORT itself, the condition should be 4 or greater.

17.1.3 The RELEASE Statement

The RELEASE statement must be executed from within the input section of your program. Each execution of the statement is responsible for providing a single record to be sorted. It is coded:

```
RELEASE record
```

where record labels an area within the data division in which you supply the record.

Once you have released the last record to be sorted you should execute an EXIT statement to return control from your input section to the sort. The System Manager will then PERFORM the output section so that you can obtain the sorted records one by one.

Note that you can terminate the sort by executing a statement:

```
EXIT WITH 4
```

which will cause exception condition 4 to be signalled by the SORT statement. This is particularly useful if you need to cancel a sort due to operator intervention or detection of an irrecoverable error.

If an irrecoverable error occurs on the work file during the input phase of the sort, an exception will be signalled by the RELEASE statement. You can trap and examine this condition by using an ON EXCEPTION statement immediately following the RELEASE statement.

Three separate conditions may be signalled:

$$COND = 11 means that an irrecoverable I/O error occurred on the work file;

$$COND = 12 means that there was insufficient space on the work file;

$$COND = 13 means that there was insufficient free memory available.

17.1.4 The RETURN Statement

The RETURN statement must be executed within the output section of your program. Each execution of the statement provides you with a single sorted record. It is coded:

```
RETURN record
```

where record labels an area within the data division in which the sort returns each record. When there are no more records remaining RETURN signals exception condition 1. You should trap this condition by means of an ON EXCEPTION statement immediately following the RETURN statement.

Following the exception, you may perform your own end-of-sort housekeeping. You must then execute an EXIT statement to return control from your output section to the sort, which will then resume the main part of your program at the instruction following the SORT statement, signalling normal completion.

If an irrecoverable I/O error occurs on the work file exception condition 11 will be signalled (not exception condition 1).

Note that you can terminate the sort by executing a statement:

```
EXIT WITH 4
```

which will cause exception condition 4 to be signalled by the SORT statement. This is particularly useful if you need to cancel a sort due to operator intervention or detection of an irrecoverable error.

17.2 Programming and Design Notes

17.2.1 Estimate of Number of Records to be Sorted

If you are unable to estimate the number of records, specify the SCRECS field as zero. The System Manager will not then check the capacity of the sort, and the SORT statement will not be liable to an exception. However, your program will be terminated in error during the sort process if the capacity is exceeded. If you do specify a positive value in SCRECS this may limit the capacity of the sort, and so you should ensure that the figure you specify is at least as large as the number of records to be sorted. Specifying the number of records may sometimes allow a sort to be performed in memory which otherwise would have to use the work file.

17.2.2 Specification of Keys

You must be careful to specify the keys correctly, as it is unlikely that any error in the specification will be detected by the System Manager.

The number and type of the keys involved affects the efficiency of the sort process. Ascending character keys are the simplest to sort, then

descending character, ascending computational and finally, most difficult of all, descending computational keys. Whenever possible, group together the keys that you need to sort frequently and sort them as a single ascending character key. Note that computational keys which cannot take on negative values may be treated as character keys.

17.2.3 Free Memory

The sort uses the free storage at the end of the user area as a workspace. The smaller the program calling the sort, the larger the workspace, and hence the larger to capacity of the sort. Therefore you may wish to make a sort and its associated processing into a separate overlay, to maximise the free storage available. Note that you may need to use the FREE$ system routine before passing control to the program to ensure that the System Manager allocates the maximum amount of free space actually available. (See the section on storage management in the Global Cobol System Subroutines Manual for more details.)

In general, additional free memory will increase the speed of a sort, particularly if this permits an in-memory sort to be performed rather than one which uses a work file.

In extreme cases, you may need to overlay the input and output section processing. You will still need skeletal, permanently resident, input and output sections, but each of these can pass control to a separate overlay introduced by a LOAD or EXEC statement. You must make sure that the input overlay is at least as large as the output overlay, and load the input overlay before initiating the sort, because the sort will use the whole of the remaining memory for its work area. When the output section is entered you must LOAD or EXEC the output overlay, which will occupy the same storage area as the input overlay. Because the top of memory is used by the sort you must set the system variable $$INDE to point at the overlay storage area, to avoid corrupting high memory. The example shows, in skeleton form, the coding required:

```
LOAD input-overlay
SORT SC SORT-IN SORT-OUT WFILE

SECTION SORT-IN
CALL $$EPT USING...  * CALL INPUT OVERLAY
EXIT            * RETURN TO SORT

SECTION SORT-OUT
POINT $$INDE AT overlay area * SET UP $$INDE
EXEC output-overlay USING... * LOAD AND ENTER
                     * OUTPUT OVERLAY
EXIT            * END SORT
```

17.2.4 File Location

For large sorts, file access time will account for most of the time spent in the sort. This time can be greatly reduced by locating the files on the fastest direct access devices available, and by putting the files on separate drive units so that the read/write heads on the drives do not have to be continually moved between the files.

The most important consideration is to avoid contention between the work file and files accessed in the output section. This can be done either by putting them on separate drives, or, if the output is

relative sequential, by using a BLOCK CONTAINS statement in the output file's FD, thus reducing the number of accesses to the output file.

17.2.5 Blocking Files

If a relative sequential file is used either as the sort input or as the sort output, then a BLOCK CONTAINS statement in the FD will normally improve performance, although it does reduce the free space available to the sort. Blocking of the output file is particularly useful if it is on the same drive as the work file. Note that it may be possible to use the same FD for both input and output files, so that they can share the same buffer area.

17.2.6 "Tag" Sorts

In a normal sort the whole of the input record is sorted. In a tag sort however, only a special, short record is passed to the sort, consisting of the sort key(s) followed by the record key. The output section of the sort must then re-read the input file, using the record key, to obtain the complete record.

The advantage of a tag sort is that because the record is much smaller the sort capacity is increased. It may be possible to perform the sort in memory instead of using a work file. The disadvantage is that each record on the input file must be read twice, and this often outweighs any time savings on the sort itself.

A tag sort must be used if a normal sort would not have sufficient capacity. A tag sort will usually be faster than a normal sort when the record constructed for the tag sort is much shorter than the original record length, particularly if this results in the sort being performed in memory. However, the performance of a sort depends critically on the machine on which it is executing and the relative performance of the two types of sort vary considerably from machine to machine.

17.2.7 Processing the Sorted Records Directly

In many cases the sorted records need not be written to a file, but can be processed immediately. For example, it is often necessary to sort records into a particular sequence simply in order to print a report. It is clearly unnecessarily time-consuming to write the records to a file and then print them in a separate program; it is better if the report is printed as part of the sort output routine. When the report is being written directly to the printer, much of the time spent in the sort will be overlapped with the printing, giving further performance benefits.

Such a sort can also be made more efficient by reducing the number and the size of the records sorted. If only certain types of records are required to produce the report, then the other records should be omitted by the sort input routine. If only a small amount of the information contained within the record is required to produce the report, it may be better to construct shorter records containing just the information required.

```
PROGRAM SORT1
DATA DIVISION
FD WFILE ORGANISATION UNDEFINED
ASSIGN TO UNIT "DSK" FILE "SORTWORK"
.
(File definitions for the unsorted input file, INFILE and the
```

```
sorted output file, OUTFILE, follow).
.
77    RECORD PIC X(100)         * RECORD AREA
01    SC                    * SORT CONTROL AREA
02 FILLER PIC 9(4) COMP
VALUE 100        * RECORD LENGTH
02 FILLER PIC 9(9) COMP
VALUE 1000            * NUMBER OF RECORDS
02 FILLER PIC 9(4) COMP
VALUE 13         * TOTAL KEY LENGTH
02 FILLER PIC 9(4) COMP
VALUE 2          * TWO KEYS
02 FILLER OCCURS 2 PIC X(8)    * KEY DESCRIPTORS:
VALUE "08XA0005"      * CUSTOMER CODE
VALUE "04CA0013"      * TRANSACTION AMOUNT

PROCEDURE DIVISION
.
(Coding to open the input file, INFILE)
.
SORT SC SORT-IN SORT-OUT WFILE
      ON EXCEPTION GO TO ERROR-REP
      .
      (If control    reaches here the   sort has completed
      satisfactorily, and OUTFILE, containing the sorted records,
      has been created)
      .
 SECTION SORT-IN
 GETINP.
      READ NEXT INFILE INTO RECORD * READ RECORD FROM INFILE
      ON EXCEPTION         * SIGNALS END OF FILE
      (Coding to close the input file)
      EXIT
      END
      RELEASE RECORD       * SUPPLY RECORD TO SORT
      ON EXCEPTION EXIT WITH $$CODE * REFLECT ERRORS TO SORT
                           ERROR HANDLING
      GO TO GETINP
 SECTION SORT-OUT
      (Coding to open the output file, OUTFILE)
 GETOUT.
      RETURN RECORD        * OBTAIN RECORD FROM SORT
      ON EXCEPTION
      IF $$COND = 1 STOP RUN * STOP IF READ ERROR ON
                           WORK FILE
      (Coding to close the output file)
      EXIT            * END OF SORT
      END
      WRITE NEXT OUTFILE FROM RECORD * OUTPUT SORTED RECORD
      GO TO GETOUT
```

Figure 13.3.1 - Sort Example 1 - A Simple Multi-key sort

17.3 Examples

In order to fit the examples onto single pages, and to emphasise the structure of the sort itself, most of the statements have been replaced by brief descriptions in parentheses. Note that the input and output files need not be relative sequential; for example, the input file in the first example could be indexed sequential.

17.3.1 Example 1 - A Simple Multi-key Sort

Figure 13.3.1 shows the skeleton of a program which uses the multi-key sort to arrange the records of an input file, INFILE, in the sequence of transaction amount within customer code. The sorted information is written to an output file, OUTFILE.

The sort control area (SC) indicates that the records involved are 100 bytes in length, that a maximum of 1000 records ever need to be sorted, and that there are 2 keys whose combined length is 13 bytes. The key description table defines the 8 character customer code, beginning at byte 5 of the record, as the most significant key. The second key, the transaction amount, is a 4-byte computational field beginning at byte 13 of the record. Both are to be sorted in ascending sequence. Note that if the amount could never be negative, these two fields could be combined into a single character key.

Before the sort takes place the input file is opened. You could use the FSTAT$ routine described in section 9.10 to determine the size of the file and deduce the exact number of records present. You could then make an accurate estimate of the number of records to be sorted, rather than just using an appropriately high value as in this example.

The SORT statement which initiates the multi-key sort identifies SORT-IN and SORT-OUT respectively as the input and output section names. A sort work file named WFILE is provided. An ON EXCEPTION statement immediately following the SORT statement routes control to ERROR-REP if insufficient space is available, or if an irrecoverable I/O error occurs when checking the work file. Control will only reach the next statement if the sort completes successfully.

The input section, SORT-IN, simply reads records one by one from INFILE and releases them to be sorted. In a more sophisticated application some of the records might well be dropped from the sort at this point, or additional records might be added. When the READ NEXT statement used to obtain records sequentially from the file signals an exception this means the end of the file has been reached. The file is then closed and an EXIT statement is executed to return control to the System Manager.

Once the records have been sorted according to the key sequences defined by the key descriptor table the output section, SORT-OUT, receives control. It extracts the records one by one using the RETURN statement and writes them sequentially to the output file. Once the last record has been supplied RETURN signals an exception. The output file is then closed and an EXIT statement executed to transfer control back to the System Manager which, in turn, passes control back to the ON EXCEPTION statement following the SORT statement. However, since processing has completed normally no exception is signalled, the ON EXCEPTION logic is skipped, and the next statement is executed.

```
PROGRAM SORT2
DATA DIVISION
.
(File definitions for input file INFIL, key INKEY, and output
fileOUTFIL).
.
01    SC                    * SORT CONTROL AREA
 02   FILLER PIC 9(4) COMP
```

```
          VALUE 13        * RECORD LENGTH
 02   FILLER PIC 9(9) COMP
          VALUE 0         * NUMBER OF RECORDS
 02   FILLER PIC 9(4) COMP
          VALUE 8         * TOTAL KEY LENGTH
 02   FILLER PIC 9(4) COMP
          VALUE 1         * ONE KEY
 02   FILLER PIC X(8)          * KEY DESCRIPTION
          VALUE "08XA0001"
*
01    RC
 02   FILLER PIC X(4)
 02   RCKEY    PIC X(8)        * SORT KEY
 02   FILLER PIC X(188)        * REST
*
01    ZA                  * RECORD FOR TAG SORT
 02   ZASKEY PIC X(8)          * SORT KEY
 02   ZARKEY PIC 9(9) COMP          * RECORD KEY
*
.
PROCEDURE DIVISION
 .
 (Coding to open input and output files)
 .
 SORT SC SORT-IN SORT-OUT
 ON EXCEPTION GO TO ERROR-REP
 .
 (If control reaches here the sort has completed successfully) (Close
input and output files and exit)
 .
SECTION SORT-IN
GETINP.
 READ NEXT INFIL INTO RC  * GET NEXT RECORD
 ON EXCEPTION EXIT
 MOVE RCKEY TO ZASKEY     * MOVE SORT KEY
 MOVE INKEY TO ZARKEY     * AND RECORD KEY TO ZA
 RELEASE ZA               * PASS ZA TO SORT
 GO TO GETINP
*
SECTION SORT-OUT
GETOUT.
 RETURN ZA                * GET NEXT KEY FROM SORT
 ON EXCEPTION EXIT
 MOVE ZARKEY TO INKEY     * MOVE KEY TO INPUT FD
 READ INFIL INTO RC       * AND READ RECORD
 WRITE NEXT OUTFIL FROM RC     * THEN WRITE TO OUTPUT
 GO TO GETOUT
```

Figure 17.3.2 - Sort Example 2 - A Tag Sort

17.3.2 Example 2 - A Tag Sort

Figure 17.3.2 shows the skeleton of a program which uses the multi-key sort to sort an input file, INFIL, using a tag sort. The sorted file is written to an output file, OUTFIL.

The record to be sorted consists of the 8 character sort key followed by a 4 byte file key. (This assumes that the file has a four byte key field, as for relative sequential files.) Hence the sort control area (SC) gives a record length of 13 bytes (8+4), with a single 8

character key starting at the first byte. Because the record length is only 13 bytes, a large number of records can be sorted in memory, typically 1000-2000, and so a work file is not required.

The input section, SORT-IN, reads records one by one from the input file, constructs the tag record to be sorted from the sort key and the record key, and releases the record to the sort.

The output section, SORT-OUT, extracts records from the sort using the RETURN statement, moves the record key to the key field of the input file definition, and then reads the original record from the input file. This record is then written to the output file.

17.4 Using the multi-phase sort, MSORT$

A new sorting subroutine is available as part of V6.2 and later System Manager which performs much the same function as the Global Cobol SORT verb. It has a number of advantages over the SORT verb, the most important being that its capacity to sort records is not limited by the amount of memory available.

MSORT$ includes a multi-phase merge mechanism, to permit it to sort unlimited amounts of data, and this mechanism is optimised for performance assuming that there are around 30 cache buffers available on the computer. (Note that on a multi-user system other programs will be competing for cache buffers, so MSORT$ will not get to use all the buffers configured.) If there are no cache buffers at all (or potentially if there are vastly more than 30) then the performance of MSORT$ may be somewhat worse than that of the SORT verb if large numbers of records are processed (although in general the performance of MSORT$ is the same as or better than that of the SORT verb).

MSORT$ also makes use of the special service used by DMAM to build its keys (this is used both to remove the key length limits on descending keys, and to provide the new translation key types), and as a consequence may not be used on a pre-V5.1 operating system. It is necessary to call the DBSET$ routine at some point in the program suite before calling MSORT$ (this is to establish the required service routine - see documentation of DBSET$ in the Data Management Manual for full details). Note: the requirement to call DBSET$ does not apply if the records to be sorted have only a single, character ascending key.

Because of the extra interface requirements of MSORT$, and possible performance degradation in certain situations, it is not currently being introduced as the standard sort invoked by the SORT verb. Nevertheless it is likely that MSORT$ will become the standard at some point in the future, and developers are encouraged to use MSORT$ in any situation where its unlimited capacity would be of value.

As with the SORT verb, MSORT$ employs the unoccupied part of the user area following the last program loaded as a work area. The size of the work area affects the efficiency of the sort, but does not limit the number of records which can be sorted.

If all the records can be sorted in memory, then no work file will be used. If a work file is required then it must be large enough to contain all the records and all the keys, and allow an expansion factor for the multi-phase merge. Details on the calculation of the expansion factor may be found in section 13.5.

17.4.1 Invoking MSORT$

MSORT$ is invoked by a call, similar to the SORT verb. You code:

```
CALL MSORT$ USING sc input output [filename]
```

Where the parameters have the same meaning as those in 13.1.1.

In addition to the key formats supported by the SORT verb, it is permissible to code a key format of T, which indicates translation. A translation key is built using a translation table held in the sort routine, which is accessible to the user program as an EXTERNAL SECTION in the following format:

```
EXTERNAL SECTION IX$TAB       * name of section
*
77   IX-NSEG   PIC 9(4) COMP  * no. of merge streams
*
01   IX-TT                         * translation table
  03 IXBITS         PIC 9 COMP          * translation type
  03 FILLER         PIC X(3)
     VALUE     LOW-VALUES
  03 IXCHAR OCCURS 256 PIC X  * translations
```

The field IX-NSEG is used to control the maximum number of input streams the sort will permit during its internal merge processing, and is set initially to 32. It is desirable to have no more streams of merge data than there are cache buffers available (for efficiency reasons), but within this limit the higher the number of streams permitted the more efficient the sort will be. The field may be amended by the user program to tune performance of the sort to the system configuration - values of less than 8 or more than 64 in IX-NSEG will be treated as 8 and 64 respectively (although future versions of MSORT$ may permit a wider range of values). If no cache is available IX-NSEG should be set to 9999 to use as many streams as possible.

The IX-TT table defines a translation table in the same format as that used by DMAM (it may be read from a DMAM database using the DBRTT$ routine documented in the Data Management manual). Consult the appropriate appendix of the Data Management manual for complete details of the table layout.

The default values in IX-TT give a RAD-50 style (3 characters into 2 bytes) compression of numerals and alphabetic characters, without regard to case and ignoring all other characters. The user program may of course establish its own translation table, either by setting the values explicitly in IX-TT, or by reading the translation table from a DMAM database file using the DBRTT$ subroutine.

When translations are in use it can be difficult to determine the total length key length of the sort, so MSORT$ calculates the precise value for this field itself. You should set SCKLEN to be the sum of the lengths of the individual keys, and the sort will establish the correct total key length in the field once it has examined all the keys and determined their correct key lengths taking translation into account. Each translation key is allocated space sufficient to hold the maximum number of bytes into which the key value could be translated.

17.4.2 Exception Conditions from MSORT$

Exception conditions 1 and 2 have the same meaning as those for the SORT verb.

Exception condition 3 indicates that there is not sufficient space in memory to hold at least 8 copies of the record to be sorted, and hence to manage at least 8 merge streams (the multi-phase merge requires a minimum of 8 streams to ensure an acceptable level of performance). In such a case you must either make more memory available as workspace, or perform a tag sort on the record in question.

17.4.3 The MREL$ routine

The MREL$ routine serves the same purpose for MSORT$ as the RELEASE statement serves in the SORT, that is to say it is used to pass records from your input routine to MSORT$ for processing. Each execution of MREL$ is used to pass a single record to MSORT$. It is coded:

```
CALL MREL$ USING record
```

Where record identifies the record to be passed to the sort.

Once you have released the last record to be sorted you should EXIT from the input routine, returning control to MSORT$ which will be performing any merging required and then call your output section so that you may obtain the sorted records.

MREL$ has the same programming interface as the RELEASE statement, and may return the same exceptions with the same meanings (except exception 13 which can only be returned when there is no workfile specified).

17.4.4 The MRET$ routine

The MRET$ routine serves the same purpose for MSORT$ to that served by the RETURN statement in the SORT, that is to say it is called from your output section to provide you with a single sorted record. It is coded:

```
CALL MRET$ USING record
```

where record identifies the area where the sorted record will be returned.

The programming interface to the MRET$ routine is the same as that of the RETURN statement.

17.5 Programming and Design Notes for MSORT$

Most of the notes in section 13.2 apply equally to MSORT$. The one that does not is the point at the end of 13.2.1, which says that specifying the number of records to be sorted may permit a sort to take place completely in memory rather than using the workfile. This is not the case for MSORT$, which never dynamically allocates table space for sorting.

17.5.1 Capacity of MSORT$

The SORT verb is limited in its capacity to sort records, in that it cannot sort more than the square of the number of records which will

fit into memory. As has already been mentioned MSORT$ has no such limitation, but the ability to sort more records means that under some circumstances MSORT$ will require a larger work file than the SORT verb.

MSORT$ can sort up to the number of records which will fit into free memory without requiring a work file at all. When there are more records than this the sort writes records to the work file in groups, each of which contains the number of records which will fit into memory. So long as there are no more groups in the work file than the number of merge streams the sort will use, then the work file needs only space to hold the records (and keys), as was the case for the SORT verb. If, however, there are more groups than the number of merge streams, then the sort will need extra space in the work file so that it can perform some merging of the groups before data is returned from the sort.

The number of merge streams the sort utilises is the minimum of the number of records which will fit into memory and the number of input streams set up in IX-NSEG (whose default value is 32). The sort will not function if there is space for fewer than 8 records in memory, nor will it handle more than 64 input streams, so these form the bounds on the number of merge streams available to the sort.

The precise amount of expansion space required by the sort to perform its multi-phase merge depends on the number of records being sorted, the size of the available free memory, and the number of merge streams available to the sort. A worst case estimate of the expansion space required can be made based purely on the number of merge streams, however, and calculations show that the expansion factor is (200%/(number of merge streams)). Thus with only 8 merge streams the worst possible case requires 25% more work file space than that taken up by the records and keys. With 32 merge streams (the default value given sufficient free memory) the worst possible expansion is around 6.5%. Clearly some consideration of the number of records which will fit into memory, and hence the number of merge streams likely to be available, should be done when producing estimates of required work file size.

17.5.2 Subroutine Size

The size of the MSORT$ routine and the subroutines used by the SORT verb are similar (the extra size of MSORT$ is taken up by pre-allocated space for holding tables of merge stream information, which would have been allocated from free memory by the SORT verb). You should try to call DBSET$ in a separate overlay to the MSORT$, however, to ensure that the maximum amount of free memory is available for sorting (DBSET$ and its included routines are quite large in total).

APPENDIX A - INDEXED SEQUENTIAL FILE STRUCTURE

Figure A1 shows how an indexed sequential file is divided into three parts: the prime data area; the index area; and the overflow area. The size of each individual area is determined when the file is first created; however, the overflow area may be extended by simply copying the file to a larger extent.

The Prime Data Area

The prime data area contains the fixed-length data records of the file from which the indexed sequential file was created. During the creation process the dummy high record is added and this, in turn, may be followed by a slack area serving to make the space allocated for prime data a multiple of 256 bytes. (This is so that the following index area begins on a 256-byte boundary within the file in order to optimise performance.)

The format of data records is shown in Figure A2. The TYPE, and USER DATA may be updated by the application program. The KEY cannot be changed: although the key length may vary from file to file it is fixed for any particular indexed sequential file when that file is created. The data records are in ascending KEY order.

The dummy high record has a TYPE of spaces, a KEY of high values, and USER DATA consisting entirely of binary zeros.

All LINK fields within prime data records (including the dummy high record) are set to -1 when the file is created to show that there are no overflow records chained. These LINK fields are updated, as explained below, when new records are added to the file.

The Index Area

The index area is built when the indexed sequential file is created from a relative sequential file, or re-organised, and is read-only from then on. The index contains a number of 256-byte blocks arranged in levels, with the first (lowest) level index at the start of the area, followed by the second level index, and so on. The highest level index, consisting of just a single block, occupies the very last block of the index area.

Each index block (see Figure A2) consists of a 4-byte file pointer and a 252-byte key table. Suppose the key table can hold n different keys. Then the first level index contains one block for every n prime data records, and holds every key appearing in the prime data, including the dummy high key. The second level index contains one block for every n blocks of the first level index, and holds only the highest key from each first level block. Index levels are constructed one after another until the highest level index, consisting of just a single block, is developed. The keys appearing within the second, third, or higher index level are always in ascending sequence. The first level index may be out of sequence if some third level entries

address it directly for optimisation purposes (explained later).

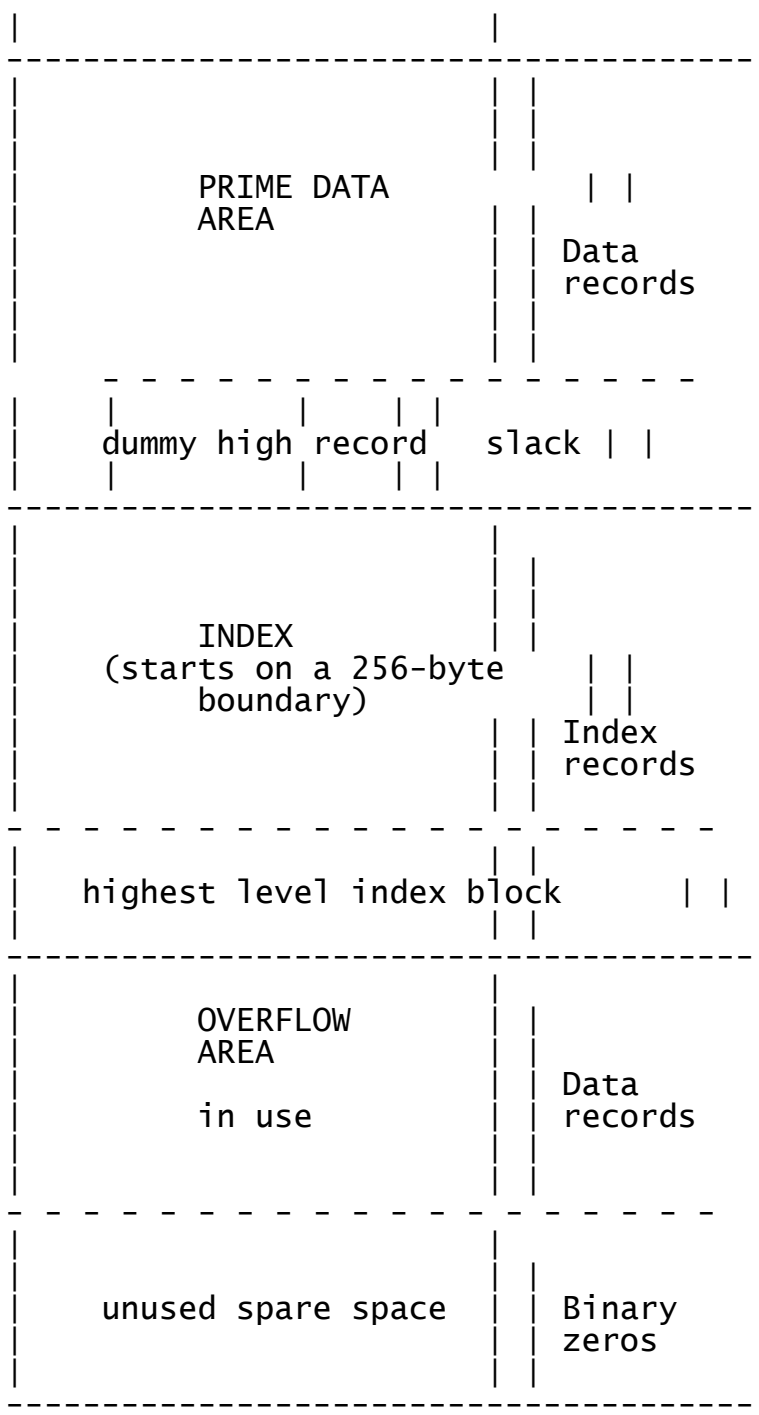


Figure A1 - Indexed Sequential File Structure

There will be unused space at the end of a key table when the key length does not divide exactly into 252. There will be additional unused space in the very last block of an index level if there are insufficient keys to fill the last key table.

In order to optimise access, the third level index can address the first level index directly for records at the beginning of the file. This results in faster access to such records when using the index, as there is no second level index to be read. Enough second level blocks are omitted to ensure that all the higher level blocks are completely full. In the first level

index, blocks which have a second level index appear before those which are accessed directly from the third level.

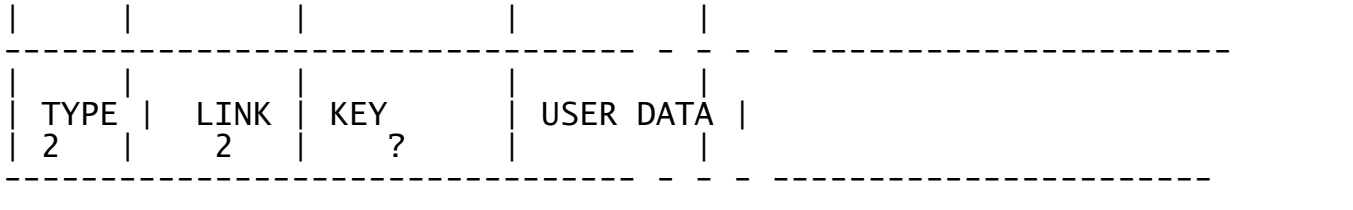
The file pointer in a first level index block addresses the data record in which the first key in the block appears, and thus the access method can use the relative position of a key within the table, together with the file pointer, to proceed directly to the data record containing a prime data key. (However, if the required key is in an overflow record an overflow chain will then have to be searched as well.) The file pointer in an index level higher than the first effectively addresses the block of the next lower level index in which the first key of the table appears as the highest key. Once again, the access method uses the relative position of the key within the table, in conjunction with the file pointer, to determine which block of the lower-level index to search. It is only ever necessary to search one block at each index level.

The Overflow Area
+++++++++++++++++
When an indexed sequential file is initially created the overflow area is empty, and each byte it contains is set to binary zeros. However, whenever a new record, containing a key which is not already present on the file, is added it is written to the first free location of the overflow area. Data records within the overflow area have exactly the same format as those within the prime data area.

The LINK field is used to place all overflow records with key values between those of two adjacent prime data area records in an overflow chain arranged in ascending key sequence. The LINK
++++++++++++++
field of the prime record with the higher key is set to point at the first record of its chain, or contains the value -1 if the prime record possesses no such chain. A LINK value of -1 in an overflow area record simply signifies that it is the last record of its chain.

A LINK value of 0 in an overflow record means that the record lies within the unused spare space. When an indexed sequential file is closed a pointer to the spare space is maintained in the file label so that the directory listing produced by the file utility indicates the amount of free storage available for over-flow records. However, to guard against the case when you fail to close the file, having inserted some records, the access method always scans the spare space to find the first free record, if any, whenever the file is opened.

DATA RECORD FORMAT
++++++++++++++++++



TYPE: A PIC X(2) code identifying the record type. If

++++
If it begins with * the record is logically deleted.

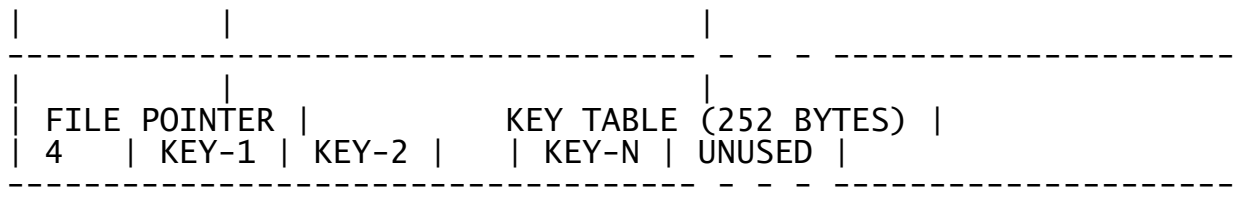
LINK: A PIC S9(4) COMP field, defined as follows:
+++++

-1 no overflow record chained

0 unused record in spare part of the overflow area

>0 record number of the chained record within the overflow area.

INDEX BLOCK FORMAT
++++++++++++++++++



FILE POINTER: A PIC S9(9) COMP field defined as follows:
++++ +++++++

Non-negative. The block is part of the lowest
++++++++++++
level index and the file pointer is the relative byte address of the data record containing KEY-1 (first record starts at byte 0).

Negative. The block is not in the lowest level
++++++++
index. The file pointer is the (negative) offset from the start of the overflow area of the index block containing KEY-1 in the next lower level of index.

Figure A2 - Data record and index block formats

Example Structure
+++++++++++++++++
Figure A3 shows the structure of an indexed sequential file after some insertions have been made. A rather artificial example has been chosen in order to keep the figure reasonably simple. In particular the keys of the file are untypically long (60 characters), so that only four of them can fit in a single index block key table.

The indexed sequential file was created from a relative sequential file containing records with key values 10, 20, 30, 40, 50, 60, 70, 80, and 90. The first level index contains just three blocks. Its last key table holds only two key values, 90 and *

(signifying the dummy high key). The second level index is also the highest level index. Its key table contains just three entries, representing the high keys from three first level index blocks.

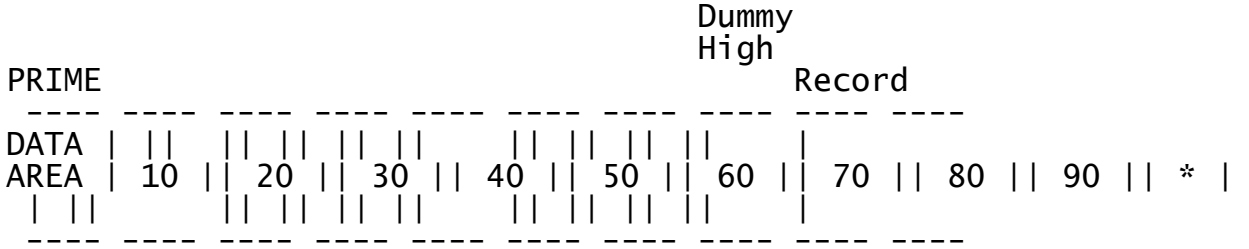
Six records have been added to the file since it was created. Their keys appear in the overflow area in the order 11, 35, 23, 21, 96 and 91, and this is the sequence in which these new records were themselves created.

Four overflow chains have been developed, as shown in the following table:

PRIME RECORD WITH KEY VALUE	ADDRESSES OVERFLOW CHAIN MADE UP OF RECORDS WITH THE INDICATED KEY VALUES
20	11
30	21 - 23
40	35
* (dummy high record)	91 - 96

Table A4 - Overflow Chains

This example illustrates the main purpose of the dummy high record, namely to control the overflow chain of records with key values higher than any appearing in the file when it was created.



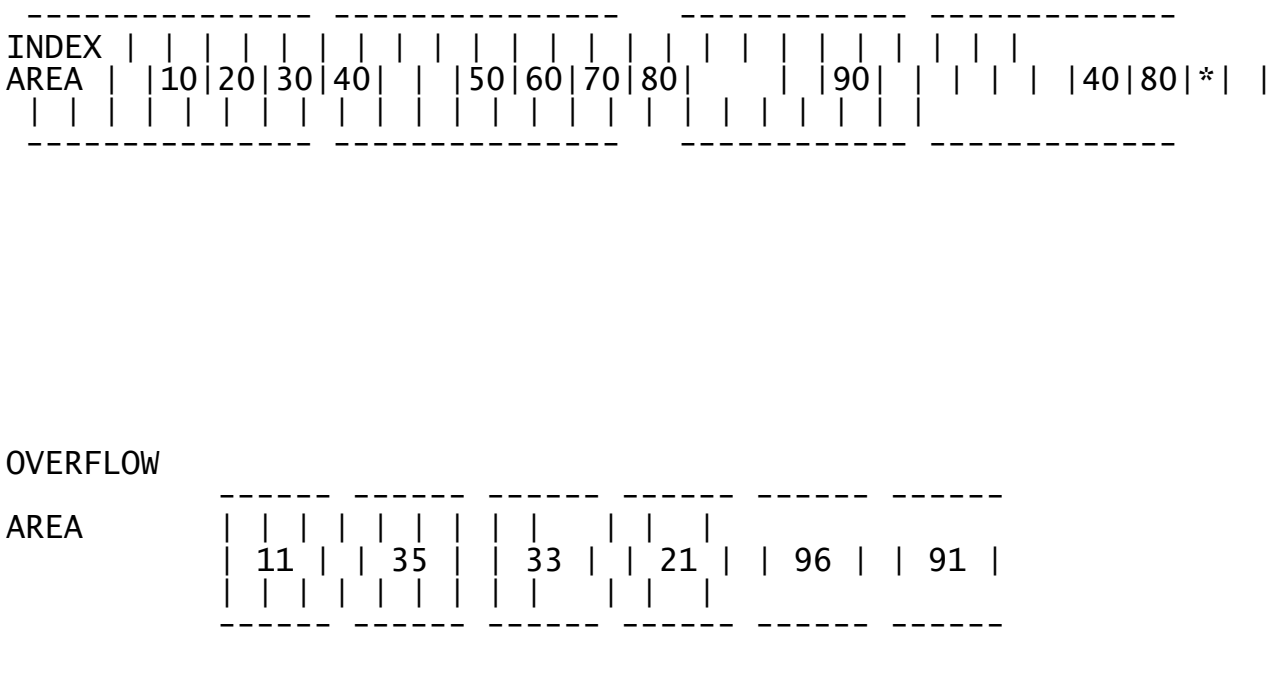


Figure A3 - Example indexed sequential file structure

File Corruption
+++++++++++++++

Rarely, hardware or software faults on a system may cause data corruption within files. If such corruption occurs within many file organisations it will be unnoticed, and the invalid data will be returned to the user program. The indexed sequential access method has a limited ability to detect file corruption however, and if it does so it will issue an appropriate STOP code, to alert the operator to this fact.

The ability of ISAM to detect file corruption falls into two broad areas. The first of these is corruption of the LINK field within the data records, and the second is general corruption of file pointers.

Where the LINK field in a record has been corrupted, ISAM can detect this if it contains a non-positive value which is not -1. Such a situation cannot arise legally, and enables ISAM to detect around 50% of random record corruption problems. In fact, as typical values for corruption are either binary zeros or large

negative values, ISAM can normally manage to detect most record corruption problems.

Where file pointers have been corrupted the problem is more complicated. Obviously a file pointer which points outside the file extent is invalid, and may be simply detected when it is used. However file pointers which spuriously point within the file may indicate apparently valid records and index blocks. In an attempt to detect these situations ISAM is careful to notice when key values lie out of sequence, for example when a lower key value is encountered when scanning an overflow chain, and in these cases too an error is indicated.

If such file corruption should occur it may well only be detected by one of a series of programs running on the same data, as detection of the corruption involves attempting to process the corrupted portions of the file. When file corruption is detected it will usually be necessary to fall back to a backup copy of the data, but it is vital to attempt to repeat the operation which
+++++
noted the corruption in the first place, as it is possible that the corruption may have occurred some time previously, and that the backup may also be corrupted.

In some situations it may be possible to reconstitute the ISAM file, with the corrupted records omitted, by using the $RECOVER utility which is documented in the Toolkit Manual. Note however that it is not, in general, possible to reconstitute ISAM files used by Global 2000 Business Software in this way. You should consult the TIS Global Hotline before attempting any recovery or reconstitution of a Business Software data file.

Appendix B - Catalogue File Structure

A catalogue file is a relative sequential file containing up to 100 nineteen-byte records. Each record represents a catalogue entry, and has the following format:

```
01   CG
  03 CGFID          PIC X(8)       * FILE-ID
  03 CGUID                      * UNIT-ID OR
    05    CGFLU     PIC S9(2)      * FLOATING UNIT
  03 CGVID     PIC X(6)        * VOLUME-ID
  03 CGDUP     PIC X                * DUPLICATE IND.
  03 FILLER         PIC X                * RESERVED
```

The field CGFID is simply the file-id corresponding to the entry. If a CGFID field in a data security catalogue begins with two asterisks, then the record it occupies represents a request to secure all the files on the volume identified by CGVID.

The field CGUID either contains a floating address, in the range -99 to 99 inclusive, or a unit-id. The two cases can be distinguished by testing if redefinition CGFLU is numeric: if it is then the entry represents a floating address. For example:

```
IF CGFLU NUMERIC
     process floating unit number
ELSE
     process unit address
END
```

The field CGVID contains the volume-id, or low-values if no volume-id checking is required (not "?"). The volume-id may contain asterisks as wild card characters, which CATA$ will replace by the equivalent characters from the volume-id of the volume actually occupied by the catalogue.

The duplicate indicator, CGDUP, is normally blank, but for catalogues used by SAVE$ and REST$, or the $SAVE and $RESTORE commands, it may contain the ' or " character that can be appended to the file-id in order to distinguish duplicate entries.

Appendix C – Using BDAM to Copy Files of Any Organisation

This appendix contains an example program showing how you can use the basic direct access method to read or create System Manager files of any organisation. The program copies any type of input file to produce an identical new output file, block by block. Of course you would normally use the COPY$ system routine to copy files on the same computer. The point of the example is to illustrate the technique to employ when implementing a communications scheme where the input and the output files are located on different System Manager computers not linked by a network.

In the example BDAM is used to copy the file defined by the INFILE FD to create a new file specified by the OUTFILE FD. The program prompts for the block size it is to use, which must be between 1 and 9999 bytes. The file label area is defined by the BDLAB field of the BD group, as explained in 6.2.9. The letters identifying the following notes also appear to the right of comment asterisks in the subsequent listing so that they can be readily correlated with the descriptions below:

A. Define BD FDs for INFILE and OUTFILE. The optional KEY statement is not required because the file will only be accessed sequentially. The SIZE statement must be coded on the input FD so that we know how many bytes (INSIZE) are to be copied.

B. Having prompted for the block size and established its value in BSIZE, use FILE$ to obtain the input file-id and unit. Then open the INFILE FD. Note that exception condition 2 will be signalled if either the file is not found, or if it is the "wrong type" for BDAM. The only type of files which cannot be processed are product files. These can only be used at a single specified installation and should not be transmitted.

C. Save the input file label in the 36-byte work field LABEL. (It contains the SIZE field and other internal information.)

D. Use FILE$ to prompt the operator for the output file-id and unit. Then move the LABEL information, defining the characteristics of the input file, to the OUTFILE FD before opening it.

E. Next open the output file, performing conventional processing if the file already exists.

F. Set the record length fields in both FDs to the common block size to be used in copying all but the last block, which may of course, be short.

G. Copy (INSIZE) bytes of INFILE to OUTFILE using the sequence:

```
READ NEXT INFILE INTO AREA
WRITE NEXT OUTFILE FROM AREA
```

repeatedly, and adjusting the record length fields appropriately before copying the last block.

H. Restore the label of the output FD to its initial status prior to OPEN by moving it in again, and then close the output file. Notice the line of code (commented out) which says:

```
MOVE 0 TO BDOCNT
```

This line is only required if you have opened the input file shared, and is needed to avoid checking in the System Manager close handling. The location of BDOCNT can be found from the BD data area on the first page of the listing.

I. Restore the label of the input FD to its initial status prior to OPEN, and then close the input file. The copy operation is now complete.

Note that any I/O error will simply terminate the example program. In a more sophisticated application OPTION ERROR would be coded in the FDs involved and error recovery logic would be added to the programs.

It is possible to open the input file shared, rather than old as in this example, but if you do this it is possible that some other user of the file may update it while you are copying it, rendering your copy invalid. You should either be satisfied that this cannot occur, or provide some suitable update locking strategy which will prevent file amendment during the copying process. Note also the special line of code, in note H, which must be activated if you are copying a shared file.

```
LISTING OF COPY FILE COPY USING BDAM       06/05/88 14.01.00 PAGE 1

 1           PROGRAM COPY
 2           *
 3           * CREATES AN OUTPUT FILE FROM AN INPUT FILE USING BDAM
 4           *
 5 0000      DATA DIVISION
 6           *
 7           ORGANISATION OR$85 TYPE 99 EXTENSION 8
 8           *
 9 0000      77    AREA  PIC X(9999)             * AREA FOR BLOCK
10 270F      77    TOMOVE PIC 9(9) COMP          * BYTES TO MOVE
11 2713      77    BSIZE       PIC 9(4) COMP           * BLOCK SIZE, 1-9999 BYTES
12 2715      77    REPLY       PIC X                   * REPLY FOR DELETE PROMPT
13 2716      77    LABEL       PIC X(36)         * WORK AREA FOR LABEL
14           *
15 273A      FD INFILE ORGANISATION OR$85        *A
16           ASSIGN TO UNIT "?" FILE INID        *A
17           RECORD LENGTH IS INLENG             *A
18           SIZE IS INSIZE               *A
19           *
20 2792      FD OUTFILE ORGANISATION OR$85             *A
21           ASSIGN TO UNIT "?" FILE OUTID             *A
22           RECORD LENGTH IS OUTLENG            *A
23           *
24 27EA      LINKAGE SECTION
25           *
26 27EA      01    BD
27 0000      02    FILLER PIC X(44)
28 002C      02    BDLAB
29 002C      03 FILLER PIC X(3)
30 002F      03 BDORG    PIC 9(2) COMP
31 0030      03 FILLER PIC X(3)
32 0033      03 BDOCNT PIC 9 COMP          *H
33 0034      03 FILLER PIC X(8)
34 003C      03 BDADE    PIC X(8)
35 0044      03 FILLER PIC X(12)
```

Sample Copy Program - Page 1

```
LISTING OF COPY FILE COPY USING BDAM       06/05/88 14.01.00 PAGE 2

 36 27EC 0 0 PROCEDURE DIVISION
 37           *
 38 27EC 0 0 SECTION MAIN
 39           *
 40 27F2 0 0       DISPLAY "BLOCK SIZE OF COPY (1-9999)"
 41 2810 0 0       ACCEPT BSIZE
 42 281C 0 0       IF BSIZE ZERO GO TO MAIN
 43 2824 0 0 IN.
 44 2824 0 0       DISPLAY "INPUT FILE"
 45 2830 0 0       CALL FILE$ USING INFILE       *B
 46 2838 0 0       OPEN OLD INFILE         *B
 47 2848 0 0       ON EXCEPTION                  *B
 48 284C 0 1       DISPLAY " NOT FOUND OR WRONG TYPE" SAMELINE
 49 2866 0 1       GO TO IN                *B
 50 286A 0 0       END                     *B
 51 286A 0 0       BASE BD AT INFILE             *C SAVE INPUT FILE LABEL
 52 2872 0 0       MOVE BDLAB TO LABEL           *C
 53 287C 0 0 OUT.
 54 287C 0 0       DISPLAY "OUTPUT FILE"         *D
 55 288A 0 0       CALL FILE$ USING OUTFILE      *D
 56 2892 0 0       BASE BD AT OUTFILE            *D COPY LABEL INFORMATION
 57 289A 0 0       MOVE LABEL TO BDLAB           *D TO THE OUTPUT FILE
 58 28A4 0 0       OPEN NEW OUTFILE        *E
 59 28B4 0 0       ON EXCEPTION                  *E
 60 28B8 0 1       DISPLAY " FILE ALREADY EXISTS - DELETE?" SAMELINE
 61 28D8 0 1       ACCEPT REPLY NULL GO TO OUT *E
 62 28E8 0 1       IF REPLY NOT = "Y" GO TO OUT*E
 63 28F2 0 1       CALL DELE$ USING OUTFILE      *E
 64 28FA 0 1       OPEN NEW OUTFILE        *E
 65 290A 0 0       END                     *E
 66 290A 0 0       MOVE BSIZE TO INLENG OUTLENG  *F SET BLOCK LENGTH
 67 291A 0 0       MOVE INSIZE TO TOMOVE         *G AND EXTENT SIZE
 68 2922 0 0       DO WHILE TOMOVE > BSIZE       *G
 69 292E 1 0       READ NEXT INFILE INTO AREA *G TRANSFER ALL BAR
 70 293E 1 0       WRITE NEXT OUTFILE FROM AREA*G THE LAST BLOCK
 71 294E 1 0       SUBTRACT BSIZE FROM TOMOVE *G
 72 295A 0 0       ENDDO                         *G
 73 295E 0 0       MOVE TOMOVE TO INLENG OUTLENG       *G TRANSFER
 74 296E 0 0       READ NEXT INFILE INTO AREA    *G LAST
 75 297E 0 0       WRITE NEXT OUTFILE FROM AREA  *G BLOCK
 76 298E 0 0       MOVE LABEL TO BDLAB           *H REFRESH LABEL AND
 77                ** MOVE 0 TO BDOCNT           *H
 78 2998 0 0       CLOSE OUTFILE                 *H CLOSE OUTPUT FILE
 79 29A8 0 0       BASE BD AT INFILE             *I
 80 29B0 0 0       MOVE LABEL TO BDLAB           *I REFRESH LABEL AND
 81 29BA 0 0       CLOSE INFILE                  *I CLOSE INPUT FILE
 82 29CA 0 0       GO TO MAIN
 83          *
 84 29CE 0 0 ENDPROG
```

Sample Copy Program - Page 2

```
LISTING OF COPY STATISTICS           06/05/88 14.01.00 PAGE 3

NUMBER OF ERRORS 0
NUMBER OF WARNINGS 0

COMPILATION OPTIONS IN FORCE :

TR - TRACE INFORMATION GENERATED IN COMPILATION FILE
NLN - NO LONG NAMES, THE FIRST 6 CHARACTERS ARE SIGNIFICANT
SD - SYMBOLIC DEBUG RECORD GENERATED IN COMPILATION FILE


PROGRAM SIZE = 29CE BYTES (HEXADECIMAL)

TOTAL NUMBER OF LINES    84 (EXCLUDING COMMENTS 71)

SOURCE FILE -     S.COPY       ON F25 CREATED 06/05/88
COMPILATION -     C.COPY       ON F05 SIZE       1.1K
LISTING FILE -    L.COPY       ON F05 SIZE       4.4K

MACHINE - KLUDGECOMP MK 3
VERSION - V6.1

COMPILATION COMPLETED
```

Sample Copy Program - Page 3

Appendix D - Included Routines

System routines referenced by the CALL statement are included in your program when it is linkage edited, as are access methods Introduced by FD statements coded in working storage. Table D overleaf shows the program names of the particular subroutines included from the system library when various language constructs described in this manual are coded. The SIZE column indicates the approximate size of each routine in bytes, rounded up to the nearest 0.1K (K = 1024 bytes). System routines described in other manuals are excluded from this table as they are listed in the appendices of the appropriate manuals.

If you require a more accurate estimate you should compile and link a program containing a GLOBAL statement for each of the required routines and file organisations. The link map will then give the total size of the included routines.

Global Cobol statement	Program name of the subroutine included	Size (Kb)
CALL ASSIG$ paged	GH$A	0.6
CALL CALC$	CY$A	0.2
CALL CATA$	BG$A + QJ$A + EC$A + BJ$A	1.6
CALL CONV$	BH$A + BE$A + EC$A	2.6
CALL COPY$	BK$A + BE$A + EC$A	1.2
CALL DELE$	BE$A + EC$A	0.4
CALL DEVIN$	OM$A + BZ$A	1.6
CALL FILE$ or FILDF$	BI$A	0.3
CALL FIX$	EN$A + EC$A	0.4
CALL FSTAT$	CI$A	0.2
CALL GTDES$ or PTDES$	BJ$A + EC$A	0.5
CALL ISUSE$	IL$A + EC$A	0.5
CALL LIBR$	BM$A + EC$A + BL$A	1.6
CALL LWORK$ or UWORK$	OA$A + ER$A	0.8
CALL MSORT$	IX$A + EI$A	4.3
CALL OPDE$	QI$A + BE$A + EC$A	0.7
CALL OPEN$ or LIST$ or CLOSE$	QW$A + EC$A	0.7

| CALL RENA$ | EB$A + EC$A | 0.5 |
| CALL REST$ | QZ$A + EZ$A + GC$A + BE$A + EC$A | 5.4 |

CALL SAVE$ or SAVEN$	QY$A + EZ$A + GC$A + QL$A + EA$A + EC$A	7.4
CALL SCR$	IH$A + EC$A + BJ$A	0.8
CALL SECUR$	CV$A	0.1
CALL SET$	CU$A	0.1
CALL SLOCK$ or SULOC$ paged	CA$A + ER$A	0.6
CALL VOLID$	QJ$A + EC$A + BJ$A	0.4
FD... ORGANISATION RELATIVE SEQUENTIAL	AR$B + EC$A	1.7
	(AR$A + EC$A)	(1.2)*
for C-ISAM	AY$Z + AZ$A + EC$A	3.1
FD... ORGANISATION INDEXED SEQUENTIAL	AI$A + AZ$A + EC$A	2.1
for C-ISAM	AY$Z + AZ$A + EC$A	3.1
FD... ORGANISATION OR$82	AP$A + EC$A	0.7
FD... ORGANISATION OR$83	AT$A + EC$A	1.2
FD... ORGANISATION OR$84	AV$A + EC$A	0.9
FD... ORGANISATION OR$85	AB$A + EC$A	0.7
FD... ORGANISATION OR$86	AL$A + EC$A + QL$A + GC$A + GA$A + EA$A	4.6
FD... ORGANISATION OR$96		
FD... ORGANISATION OR$97		
FD... ORGANISATION OR$98		
FD... ORGANISATION OR$99R	AS$R + CA$A + ER$A + EC$A	4.0
FD... ORGANISATION OR$99C		
SORT or RELEASE or RETURN	QO$A + QN$A + EI$A	3.9
LOCK or UNLOCK	ER$A	0.4

* Smaller size applies if BLOCK CONTAINS statement not used

Table D - Included Routines