2200/VS LCO
PROGRAMMER'S REFERENCE GUIDE

First Edition -- September 1986

715-0562

19200 D O X O O

2400 8 X X X O

1200 6 X O O X

19.200 ↓ ↑ ↓ ↑ ↑

2400 ↓ ↑ ↑ ↑ ↓

1200 ↓ ↑ ↓ ↑ ↓

1 5
O X O X X
↓ ↑ ↓ ↑ ↑

2400 O X X X O

1200 O X O X O

Draft Copy Confidential

PREFACE

The Wang 2200/VS Local Communications Option (LCO) (Version 1.0) enables Wang 2200 users to emulate up to four VS Workstations and store and access files on a connected VS system. This document provides information on programming requirements for accessing the Wang 2200/VS LCO filing services. The primary audience for this document is the Wang 2200 programmer. Experience with programming on a Wang 2200 system is recommended for the all users.

Chapter 1 provides background information on the Wang 2200/VS LCO and provides an introduction to the rest of the guide. Chapter 2 explains programming requirements for accessing VDISK files (2200 disk image files) stored on a VS system. Chapter 3 explains programming requirements for accessing native DMS files stored on a VS.

Additional information on the Wang 2200/VS LCO is available in the following Wang publications:

- 2200/VS Local Communications Options User's Guide (715-0564)
- 2200 BASIC-2 Disk Reference Manual (700-4081)
- 2200 BASIC-2 Error Codes Booklet (700-7170)
- 2200 BASIC-2 Language Reference Manual (700-4080)
- 2200 Introductory Guide (700-4613)
- VS Data Mangement System Reference (800-1124-01)

CONTENTS

CHAPTER 1 PROGRAMMER"S INTRODUCTION TO THE 2200 LCO

CHAPTER 2 2200 VDISK ACCESS

CHAPTER 3 INTRODUCTION TO NATIVE DMS ACCESS

CONTENTS (continued)

CONTENTS (continued)

APPENDIX A ADDITIONAL INFORMATION

TABLES

CHAPTER 1
PROGRAMMER'S INTRODUCTION TO THE 2200 LCO

1.1 OVERVIEW

The 2200/VS Local Communications Option (LCO) is a data communications hardware (the 2258 controller) and software option that enables a Wang 2200MVP, -LVP, or Micro-VP system to communicate with a Wang VS computer system. The 2200/VS LCO enables you to perform the following functions:

- Log on to the VS and run VS application programs that do not require the downloading of microcode to a VS workstation.
- Run 2200 application programs that store data to and retrieve data from 2200 disk-image files (VDISKs) stored on the VS system.
- Run 2200 application programs that store data to and retrieve data from native VS DMS files, using subroutines provided with the 2200/VS LCO package.

This programmer's reference guide explains the programming requirements for accessing VDISKs and native DMS files stored on the VS system with new and existing programs. For more information on using VS workstation emulation and file services utilities, refer to the 2200/VS Local Communications Option User's Guide.

1.2 2200/VS LCO FILING SERVICES

The Filing Services component of the 2200/VS LCO package enables you (a 2200 user) to access (create, read, and write) VS Data Management System (DMS) files through existing and new 2200 BASIC-2 application programs. The VS DMS files are stored on an attached VS system. The VS Data Management System manages all disk file space and services all file I/O requests on the VS system.

The 2200 LCO enables you to store and access information on a VS system in the following forms:

- VDISK
- Native VS DMS files

VDISKs are 2200 disk-image files that are stored on a VS system connected to a 2200 system through the 2200/VS LCO. A VDISK acts like a disk platter to the attached 2200 and enables you to access information stored at the VDISK address as records, files, or sectors.

You create VDISKs on the VS using utilities provided with the 2200/VS software. VDISKs can be shared with other 2200 systems equipped with the 2200/VS LCO package. VDISKs can also be accessed by the VS system. However, you would have to alter the VS application programs in order to use the files stored on VDISK. How you access VDISKs from an attached 2200 is explained in Chapter 2 of this guide.

Native VS DMS files are files that are formatted by the VS DMS (Data Management System). DMS files services supports several different file types, including: consecutive, relative, and indexed files. Native DMS files can be accessed both by VS application programs and by other 2200 systems that are attached to the VS and equipped with the 2200/VS LCO package.

To access Native DMS files you must use the DMS Access Subroutines included with the 2200/VS LCO package. Chapter 3 of this guide explains how you access Native DMS files.

CHAPTER 2
2200 VDISK ACCESS

2.1 INTRODUCTION

Using the 2200 LCO to access information stored on an attached VS system through a VDISK is just like using any other 2200 disk facility.

First you must create the VDISKs and assign them disk addresses using the File Services utilities that come with the 2200/VS LCO software. For more information on the VDISK File Services Utilities, refer to the 2200/VS Local Communications Options User's Guide.

Once you create a 2200 VDISK on the VS, you can use it like any other formatted disk with existing or new 2200 BASIC-2 application programs to write data to and read data from the VDISK.

VDISKs can be shared with other 2200 systems that are equipped with the 2200/VS LCO package. VDISKS can also be accessed by VS application programs, however it is not recommended.

When the VDISKs are opened, any user on an attached 2200 can run an application program and access the disks. The 2258 controller responds to the disk address it receives from the application program and passes filing requests to the VS Filing Services program. When the application program is finished processing, you can then run another 2200 application program.

2.2 USING VDISKS

You can use VDISKs with both new and existing BASIC-2 application programs. To access VDISK through new or existing programs you must know which VDISKs are available. You can use the View VDISK function (included in the 2200/VS File Services Utilities) to view a list of available VDISKs. Once you know which VDISKs are available, you can alter existing programs (if necessary) or create new ones to access the VDISKs.

2.2.1 Using VDISK With Existing 2200 Programs

Using VDISKs with existing 2200 BASIC-2 programs should require little or no change to the application programs, except in the following cases:

- If the disk address is hard-coded in the BASIC-2 application program.
- If the BASIC-2 application program and its data file share the same disk address.
- If the program contains a disk address verification program that does not recognize the VDISK addressing scheme.

What To Do When the Disk Address is Hard-coded

If the disk address is hard-coded in the application program you can go through the program and change all references to the disk address to the new VDISK address.

For example, if the disk address was hard-coded in the program as follows:

```
10 SELECT #5 320
```

You could go through the program and change every occurrence of 320 to an existing VDISK address. For example, the example above could be written as follows:

```
10 SELECT #5 D31
```

In this example, 320 has been changed to D31 specifying a valid VDISK address.

You can also substitute a variable for the disk address and have the program request the address from the user at run-time. You could then assign the value received from the user to the variable.

What To Do When the Program and Data Files Share the Same Disk

When the program and data files are stored on the same disk it might not be advisable to use VDISK for performance reasons (see the section on VDISK Performance Considerations in this chapter).

However, if the data file is sufficiently large to warrant the use of VDISK, you could follow the suggestions in the previous section and hard-code the changes or request the user to input the disk address. In this case, it would be advisable to separate the application program from the data file and only store the data files on the VDISK.

For more information refer to the appropriate BASIC-2 reference and disk manuals.

What To Do When Verification Procedures Do Not Recognize the Address

If a program contains a disk verification routine that does not recognize the disk addressing scheme used for VDISK, the verification routine should be updated to allow valid VDISK addresses. The valid ranges of addresses for VDISK are as follows:

- DX0 through DXF
- DY0 through DYF

The possible values for X are 1, 2, or 3. You set the value for X in the 2258 controller itself. The value of Y is dependent on the value you assigned to X as follows:

- If X is 1 then Y must be 5
- If X is 2 then Y must be 6
- If X is 3 then Y must be 7

For more information on how to set the address, refer to the 2200/VS Local Communications Options User's Guide.

2.2.2 How to Submit Programs that Access VDISK

Before you can access a file stored on VDISK, you must make certain that the following procedures have been implemented:

- That the VDISK has been created using the VDISK utilities.

- That the 2200 is actively connected to the VS (that the attach procedure has been run).

- That the program you submit has or requests a valid VDISK address as explained in this chapter.

For more information on how to create VDISKs and how to attach to run the attach progam, refer to the 2200/VS Local Communications Options User's Guide. Once you have implemented these procedures, you can submit programs tO access VDISK.

2.3 VDISK PERFORMANCE CONSIDERATIONS

VDISK performance is strongly affected by the following components of the 2200 LCO software package:

- The 2258 firmware
- The VS serial I/O processor

The 2258 firmware can handle up to four separate tasks, however, only one of these tasks can be assigned to VS Filing Services. You attach to VS Filing Services as described in the 2200/VS Local Communications Options User's Guide. Once the VS Filing Services task is active, all 2200 partitions (up to 16) can invoke VS Filing Services through the task.

It is important to remember that the 2258 controller can receive requests from any of the 16 possible 2200 partitions and that these requests are handled similarly to 2200 disk requests. As a result, the more requests that are channeled to the VS through a single 2258 controller, the slower the response time experienced by each individual partition.

The VS serial I/O processor (SIOP) handles requests in a serial manner, one request at a time. Each file request requires both a transmission to the SIOP and a response from the SIOP. Since the SIOP handles these requests in a serial manner and since the 2258 must handle requests for all partitions, VDISK performance can be adversely affected by the number of requests being processed at any given time.

VDISK performance can also be adversely affected by opening VDISKs in shared mode. Since VDISKs are actually VS files, response time can be reduced by the additional overhead in the VS system associated with shared files.

When you open VDISKs in exclusive mode, the disks are open to all partitions on the same 2200. You should only open VDISKs in shared mode when the VDISK must be shared with another 2200, or when it must be accessed through another 2258 controller on the same 2200.

Disk commands execute more slowly as the number of VDISKs opened in shared mode increases. When a disk command is executed, the 2258 firmware uses VS DMS commands to lock each VDISK opened in shared mode. The VDISKs are locked to prevent other 2258 controllers from accessing the VDISK.

If there are many VDISKs open in shared mode, the locking procedure can add significantly to the processing time required to execute each disk command.

2.4 IMPROVING VDISK PERFORMANCE

You can implement the following recommendations as required to improve VDISK performance:

- Use VDISK only to store large data files for data intensive programs and for backup storage of program files.

- Do not load programs off VDISK, particularly programs that use program overlays.

- Do not use VDISK to store files required by programs such as screen or message files.

- Avoid opening VDISKs in shared mode. However, if you must use VDISKs in shared mode, performance might be improved by bracketing each string of disk commands with a $OPEN and a $CLOSE. Bracketing a string of disk commands with a $OPEN and a $CLOSE reduces processing by the 2258 controller for files opened in shared mode.

- In large systems or where heavy use of the 2258 data link is expected, using multiple 2258 controllers for VDISK access might increase throughput proportionally. By adding additional 2258 controllers to a single system you can off-load some of the traffic to the additional controllers.

  Note that because of the 2200 disk addressing scheme you are limited to a maximum of three 2258 controllers on a single system for VDISK purposes. You can, however, add additional 2258 controllers for either native DMS access or to add more terminals for VS Workstation Emulation. Adding additional 2258 controllers for these purposes might also improve VDISK performance in large systems.

- The 2258 support utilities enables you to define multiple VDISK maps that assign 2200 platter addresses to an equivalent number of 2200 disk image files. In certain cases you might be able to improve performance by picking a specific VDISK map for a particular application. Once the application is run, you can then return the VDISK map to your normal processing configuration. However, it is recommended that you maintain one VDISK map throughout each session whenever possible.

2.5 MOVING EXISTING 2200 FILES TO VDISK

To move existing 2200 files to VDISK, use the Move Files Utility. You access the Move Files Utility from the System Utilities Menu. For more information on how to use the Move Files Utility, refer to the 2200 Introductory Guide.

CHAPTER 3
INTRODUCTION TO NATIVE DMS ACCESS

3.1 INTRODUCTION

The 2200/VS LCO package enables you to create and access files on the attached VS system directly from your BASIC-2 application programs. To create and access Native DMS files using the 2200/VS LCO, you must use the DMS Access Subroutines that come with the 2200 LCO software. Files created in this manner are referred to as native DMS files because they are managed by the VS DMS.

Native DMS files can be accessed by BASIC-2 application programs on any 2200 system attached to the VS and equipped with the 2200/VS LCO package. Native DMS files can also be accessed by VS application programs.

The 2200 LCO software supports the following DMS file types:

Consecutive

Allows you to access records sequentially, and read records on disk directly by record sequence number. Records can only be added at the end of the file, and cannot be deleted. This structure is appropriate for most data entry and batch update applications. Consecutive files are supported for all types of I/O devices and are used for specialized puposes, such as printer files and system-maintained journals.

Indexed

Allows you to access records through a key field that contains unique data values. Indexed files can only be created and stored on disk storage devices. This structure supports sequential record retrieval, and rapid non-sequential retrieval of single records from disk files by key value. You can add, update, or delete records by specifying the primary key value of the desired record.

DMS supports both primary key and alternate key indexed files.

Relative

Relative files contain sequential, fixed length record slots. RElative files can only be created and accessed on disk storage devices. Relative files allow you to access records either sequentially or directly by record sequence number. You can add, update, or delete records within a relative file. However, you must preallocate space for adding records; deleting records does not reduce the size of the file. You should choose a relative file structure if speed of access and ability to modify and delete existing records is a major consideration. Relative files are not supported on the VS-50 or VS-80 computers.

NOTE

Relative files cannot be opened in shared mode.

For more information about DMS file structures, refer to the VS Data Mangement System Reference guide.

3.2 BRIEF DESCRIPTION OF THE DMS ACCESS SUBROUTINES

Table 3-1 lists the DMS Access Subroutines available, provides a description of each subroutine and lists the subroutine's corresponding function number.

Table 3-1. DMS Access Subroutines

Function	Description	Function Number
GENERAL OPEN	Opens any DMS file.	'101
GENERAL CLOSE	Closes any DMS file.	'102
CONSECUTIVE READ	Reads a consecutive record.	'103
CONSECUTIVE WRITE	Writes a consecutive record.	'104
CONSECUTIVE REWRITE	Rewrites a consecutive record.	'105
CONSECUTIVE SKIP	Skips a specified number of consecutive records.	'106
CONSECUTIVE LOCK	Locks a consecutive file.	'107

(continued)

Table 3-1. DMS Access Subroutines (continued)

Function	Description	Function Number
CONSECUTIVE UNLOCK	Unlocks a consecutive file.	'108
INDEXED READ	Reads an indexed file.	'109
INDEXED READ NEXT	Reads the next record in an indexed file.	'110
INDEXED WRITE	Writes to an indexed file.	'111
INDEXED REWRITE	Rewrites an indexed record to a file.	'112
INDEXED DELETE	Deletes an indexed record from a file.	'113
INDEXED FIND	Finds a specified indexed record in and indexed file.	'114
INDEXED LOCK	Locks an indexed file.	'115
INDEXED UNLOCK	Unlocks an indexed file.	'116
RELATIVE READ	Reads a record form a relative file.	'117
RELATIVE WRITE	Writes a record to a relative file.	'118
RELATIVE REWRITE	Rewrites a record to a relative file.	'119
RELATIVE DELETE	Deletes a relative record from a relative file.	'120
BLOCK READ	Reads a block of data from a block file.	'121
BLOCK WRITE	Writes a block of data to a block file.	'122

The DMS Access Subroutines also provide functions for creating, deleting and renaming file and for getting file attributes. These subroutines are called the DMS Catalog Functions. Table 3-2 provides a brief description of these functions.

Table 3-2. DMS Catalog Functions

Function	Description	Function Number
FILE CREATE	Creates a DMS file.	'200
FILE DELETE	Deletes a DMS file.	'201
FILE RENAME	Renames a DMS file.	'202
GET FILE ATTRIBUTES	Retrieves the value of one or more attributes groups associated with the opened file	'203

Chapter 4 of this guide provides a detailed description of the DMS Access Subroutines and DMS Catalog functions listed above.

3.3 HOW TO USE THE DMS ACCESS SUBROUTINES

The DMS Access Subroutines are stored in the following files that are included with the 2200 LCO software:

- VSACESS0 This file contains the GENERAL OPEN, and GENERAL CLOSE subroutines. This file also includes the subroutine used to communicate all requests to the 2200 LCO controller.
- VSACESS1 This file contains all the subroutines that deal with consecutive DMS files.
- VSACESS2 This file contains all the subroutines that deal with indexed DMS files.
- VSACESS3 This file contains all the subroutines that deal with relative DMS files.
- VSACESS4 This file contains all the subroutines that enable you to access DMS files in block mode.
- VSACESS9 This file contains the DMS Catalog Functions.

To use the DMS Access Subroutines, perform the following steps:

1. Copy the files containing the required DMS Access Subroutines into your BASIC-2 application program. Usually you will be required to copy VSACESS0 and one other file into your program to open and close files and to handle all other file processing.

NOTE

When you copy the DMS Access Subroutines into your program, be certain they do not overlay lines of code in your program.

---

2. Once you have copied the DMS Access Subroutines into your program, you access the subroutines by writing a GOSUB ' to the specific function you want to perform.

For example, if you were working with existing consecutive files, you would copy VSACESS0 and VSACESS1 into your program. The subroutines in VSACESS0 allow you to open and close. The subroutines in VSACESS1 allow you to perform other functions such as reading and writing to and from the file.

The following statement is an example of how you would code a GOSUB ' to perform a GENERAL OPEN:

```
0100 GOSUB '101 (N$, T$, M$)
```

The variables (N$, T$, and M$) pass the name and organization of the file and the mode the file is to be opened in to the subroutine.

3.4 DMS ACCESS PROGRAMMING REQUIREMENTS AND PERFORMANCE CONSIDERATIONS

To use the DMS Access Subroutines, you must copy the required files into your program.

To save space, only copy the files required by the program. For example, if your program only uses existing indexed files, you only need to copy in VSACESS0 and VSACESS2. If your program uses more than one file type, you will have to copy more modules into your program.

If your program also creates, renames or deletes files, you will also have to copy in the DMS Catalog functions VSACESS9.

Once you have copied the DMS Access Subroutines into your program, you can save the program with the SR parameter. Saving the program with the SR parameter removes the REM statements from the program, thus saving you partition space.

3.4.1 How to Submit Programs that Access Native DMS Files

Before you can access a native DMS file, you must make certain that the following procedures have been implemented:

- That the 2200 is actively connected to the VS (that the attach procedure has been run).

- That the program contains the required DMS Access Subroutines.

For more information on how to attach to run the attach progam, refer to the 2200/VS Local Communications Options User's Guide. Once you have implemented these procedures, you can submit programs from the 2200 to access native DMS files stored on the VS.

3.5 GENERAL NOTES ON THE DMS ACCESS SUBROUTINES

This section provides background on and general information common to all the DMS Access Subroutines.

3.5.1 Variables Reserved by the DMS Access Subroutines

All the variables used by the DMS Access Subroutines start with the letter V. If you have any variables in your programs that begin with the letter V, you can either change the variable in your program or change the variable in the DMS Access Subroutine.

Table 3-3 lists the variables used by the DMS Access Subroutines and gives a brief explanation of their purpose.

Table 3-3. Variables Used by the DMS Access Subroutines

Variable Name	Description	Length (in bytes)
V$	Holds the file name.	32
V0	Is a work variable for the CONSECUTIVE SKIP subroutine. Holds the number of records to be skipped.	8
V0$	Holds the return code. Return codes are explained in Appendix A.	2
V1	Is a work variable.	8
V1$	Holds the file organization identifier.	1
V2	Is a work variable.	8
V2$	Holds the open mode identifier.	1
V3	Holds the key position for indexed files.	8
V3$	Holds the hold option identifier.	1
V4	Holds the key path for indexed	8
V4$	Holds the alternate mask for indexed files.	2
V5	Holds the key length for indexed files.	8
V5$	Holds the search criteria for the INDEXED FIND subroutine.	2
V6$	Holds the key value for indexed files.	6

(continued)

Table 3-3. Variables Used by the DMS Access Subroutines (continued)

Variable Name	Description	Length (in bytes)
V7	Holds the number of records to read.	8
V7$	Holds the Extended File Sharing (EFS) header.	32
V8$	Is a work scalar variable.	16
V8$()	Is a work array.	256
V9$	Holds the file identifier number.	2
V9$()	Is the data buffer array for reading in information form the file.	4096

3.5.2 Using Variables or Literals for Input Parameters

The DMS Access Subroutines enable you to pass input parameters in the form of variables or literals. For example, a GOSUB ' to perform a CONSECUTIVE WRITE can be write as follows:

```
3000 DIM H1$32,D$50,H$2
3005 H$=V9$: V9$ is the file identifier received from the OPEN
3010 H1$ = V7$:REM V7$ is the EFS header received from the OPEN
3020 D$ = "This statement is written to the file as a record."
3030 GOSUB '104 (H$, H1$, D$)
```

This statement also could have been written with a literal for D$ as follows:

```
3000 GOSUB '104 (H$, V7$, "This is written to the file as one record.")
```

In this example a literal is used for the data buffer parameter.

NOTE

In the example above, the DMS Access Subroutine variable (V7$), which stores the External File Sharing (EFS) information is used to specify this information in the GOSUB ' call to the CONSECUTIVE WRITE subroutine. You can only use the DMS Access Subroutines variables within your GOSUB ' statements if your program only has one file open at a time. Otherwise, you must assign the values stored in the DMS Access Subroutine variables to other variables within your program, and pass the appropriate variables to the DMS Access Subroutines.

3.6 MOVING EXISTING FILES FROM THE 2200 TO THE VS

The 2200/VS LCO software package provides you with the facility to move files from the 2200 to the VS. To accomplish this task, you need to write a utility program to perform the following tasks:

1. Open the 2200 file.

2. If the file does not exist on the VS, you must create the file on the VS. To create a file on the VS, you can include the CREATE FILE subroutine in your utility program, or you can use the CREATE utility provided on the VS. For more information, refer to the detailed description of the CREATE FILE subroutine in Chapter 4 of this guide.

   If the file already exists on the VS, you must use the GENERAL OPEN (GOSUB '101) DMS Access Subroutine to open the file.

2. Read a record from the 2200 file using 2200 disk access statements.

4. Write a record to the VS using the appropriate DMS Access Subroutine.

5. Repeat steps 3 and 4 until all the records are written to the VS file.

6. Close the VS file using the GENERAL CLOSE DMS Access Subroutine (GOSUB '102).

7. Close the 2200 file.

Once you have written the utility program, you must follow the procedures for submitting programs to access DMS files provided in this chapter.

3.7 MOVING FILES FROM THE VS TO THE 2200

The 2200/VS LCO software package provides you with the faacility to move files from the VS to the 2200. To accomplish this task, you need o write a utility program to perform the following tasks:

1. Open the DMS file on the VS using the GENERAL OPEN (GOSUB '101) DMS Access Subroutine.

2. Read the file using the appropriate DMS Access Subroutine (either consecutive, indexed, relative, or block).

3. If the file does not exist on the 2200 system, you can use the DATA SAVE DC OPEN statement.

   If the file exists on the 2200 system, you can use the DATA LOAD DC OPEN statement.

4. Write the file to the 2200 using the familiar 2200 BASIC-2 statements.

Once you have written the utility program, you must follow the procedures for submitting programs to access DMS files provided in this chapter.

CHAPTER 4
DETAILED DESCRIPTION OF THE DMS ACCESS SUBROUTINES

4.1 INTRODUCTION

This section gives a detailed description of the DMS Access Subroutines. The description of each subroutine includes the following information:

- General Form Section: This section shows the general format of the GOSUB' statement used and includes a description of the required input parameters.

- Purpose Section: This section explains the function the subroutine performs.

- Returns Section: This section explains information returned by the subroutine.

- Example: This section provides an example of how the subroutine is used.

A General Notes section is also included for certain subroutines to provide additional information.

4.2 NOTES ON THE GENERAL FORM SECTION

In the General Form section, the following basic rules of syntax are followed.

1. The following symbols must be included in your BASIC-2 statements exactly as they appear in the General Form of the statement:

• Uppercase letters	A through Z
• Comma	,
• Double Quotation Marks	"
• Parentheses	()
• Pound Sign	#
• Slash	/

2. Lower case letters and words in the General Form of a statement represent items whose values must be assigned by the programmer. For example, if the lowercase word "name" appears in a General Form, the programmer must substitute a specific file name (such as "PROG1"), or an alphanumeric variable containing the name, in the actual statement. Similarly, where the lowercase letter n appears, the programmer must substitute an actual file number (from 0 to 64) or a variable containing a file number.

3. All information that appears between parentheses must be included in the GOSUB' statements.

4. Blanks (spaces) are used to improve readability and are meaningless.

5. The sequence the terms are listed in must be followed.

GENERAL OPEN

General Form:

```
GOSUB '101 (file-name , org , mode)
```

Where:

file-name — is the name of the file. The file name can include the //SYSTEM/VOLUME/LIBRARY/FILENAME. The file name can also be written as ///VOLUME/LIBRARY/FILENAME. The SYSTEM, LIBRARY, and FILENAME can each be up to 8 characters in length. The VOLUME can be up to 6 characters in length.

org — is the organization of the file. The following lists the valid file organization parameters and their meaning:

- C indicates the file is a consecutive file.
- I indicates the file is an indexed file.
- R indicates the file is a relative file.
- B indicates the file can be accessed in block mode.

mode — is the mode the file is opened in. The following lists the valid mode parameters and their meaning:

- R indicates the file is opened for read only access.
- S indicates the file is opened for shared access.
- X indicates the file is opened for exclusive access.
- E indicates the file is opened for extended access.

PURPOSE:

The GENERAL OPEN ('101) subroutine enables you to open any DMS file. The subroutine enables you to specify the file name, including the system, the library, and actual name of the file. The library, volume and file name are required; the system name is optional. The GENERAL OPEN subroutine also enables you to specify the file type (Indexed, Consecutive, or Relative) and the access mode (Read Only, Shared, Exclusive, or Extended.)

NOTE

Relative files cannot be opened in shared mode

RETURNS:

The GENERAL OPEN subroutine returns the following information:

- V0$ is the return code. Refer to Appendix A for more information.
- V1 is the length of valid data in V9$() array (the file data field).
- V7$ is the EFS header information for the file indicated by V9$ (the file identifier). Refer to Appendix A for more information.
- V9$ is the file identifier assigned to the file.
- V9$() is an array of 64 or 75 bytes that contains file attribute information. Refer to Appendix A for more information.

EXAMPLE: (GENERAL OPEN)

```
10 DIM N$32,T$1,M$1
20 N$="//SYSNAM/ANYVOL/ANYLIB/FILENAME": T$="I", M$="S"
30 GOSUB '101 (N$, T$, M$)
40 IF V0$=HEX(FF) THEN 60
50 STOP "ERROR IN OPEN"
60 H$=V9$: REM V9$ is the file identifier
```

In this example variables are used to represent the name, org, and mode parameters. This example could have been written as follows:

```
10 GOSUB '101 ("///ANYVOL/ANYLIB/FILENAME","I", "S")
20 IF V0$=HEX(FF) THEN 40
30 STOP "ERROR IN OPEN"
40 REM Good general open. Continue processing.
50 H$=V9$: REM V9$ is the file identifier
```

In the examples above, the specified files are opened. In the second example, the system name is replaced with a slash (/).

General Form:

GOSUB '102 (file-id , efs)

Where:

file-id	is an alpha-numeric variable that represents the file identifier assigned to the file.
efs	is an alpha-numeric variable that represents the EFS header information for the specified file.

PURPOSE:

The GENERAL CLOSE ('102) subroutine enables you to close any DMS file that had been previously opened for I/O functions. You must specify the file identifier assigned to the file.

Attempting to close a file that was not previously opened by an OPEN statement, causes a recoverable program error at run time.

RETURNS:

The GENERAL CLOSE subroutine returns the following information:

- V0$ is the return code. Refer to Appendix A for more information.
- V1 is the length of valid data in V9$() array (the file data field).
- V7$ is the EFS header information for the file indicated by V9$ (the file identifier). Refer to Appendix A for more information.
- V9$() is an array containing 3 bytes and should be HEX(00).

EXAMPLE: (GENERAL CLOSE)

```
10 H1$=V7$:REM V7$ is the EFS header.
20 H$=V9$: REM V9$ is the file identifier
30 GOSUB '102 (H$,H1$)
40 IF V0$=HEX(FF) THEN 60
50 STOP "ERROR IN CLOSE"
60 ...
```

In this example a variable (H$) is used to represent the file identifier and the EFS information (H1$).

General Form:

```
GOSUB '103 (file-id , efs , hold , time)
```

Where:

file-id — is an alpha-numeric variable that represents the file identifier assigned to the file.

efs — is an alpha-numeric variable that represents the EFS information for the specified file.

hold — indicates the hold option and can be either "H" or " ". "H" holds the record for exclusive processing. " " allows other programs to access the record.

time — is the amount of time in seconds the system will wait if the record is being held by another user. the time parameter must be zero, unless the file is opened in shared mode. Specifying a value of zero indicates that the system will wait indefinitly.

PURPOSE:

The CONSECUTIVE READ ('103) subroutine enables you to read the next consecutive record in a specified file. The file must have previously been opened.

RETURNS:

The CONSECUTIVE READ subroutine returns the following information:

- V0$ is the return code. Refer to Appendix A for more information.
- V7$ is the EFS header information for the file indicated by V9$ (the file identifier). Refer to Appendix A for more information.
- V9$() contains eight bytes of status information and the data read from the file. Refer to the CONSECUTIVE READ General Notes section for more information.

GENERAL NOTES

In the CONSECUTIVE READ subroutine, the V9$() array is used to store status information and the data read from the file. The first eight bytes of the array are used to store the status information. Table 4-1 describes the status information.

Table 4-1. CONSECUTIVE READ Status Information

Byte(s)	Description
01	Contains internal processing information. This value should always be equal to 01.
02 and 03	Contain the number of records read.
04 and 05	Contain the number of bytes in the byte block.
06	Contains the datablock ID. This value should always be equal to 01.
07 and 08	Contains the number of bytes read. The value o fthese two bytes can be up to 64K.

The remaining bytes of the array (starting at the ninth byte) are used to store the data read from the file. If you know the length of the records read, you will know how many bytes of V9$() will be used to store the data. If you do not know the length of the records in the file, or the file contains variable length records you can use the VAL function on the 7th and 8th bytes of V9$() array to get the length of the data read. See the Example section for more information.

The V9$() array is originally dimensioned to hold 4096 bytes of information (including the status information). However, you can decrease the size of the array depending on your needs.

EXAMPLE: (CONSECUTIVE READ)

The following is an example of the CONSECUTIVE READ subroutine:

```
10 DIM C1$5,N1$30,B1$3,W$8
20 H$=V9$:H1$=V7$:T=25
30 W$=HEX(A005A01EA0035205)
40 GOSUB '103 (H$, H1$, "H", T)
40 IF V0$=HEX(FF) THEN 60 ELSE 50
50 STOP "ERROR IN READ"
60 $UNPACK (F=W$) STR(V9$(),9,VAL(STR(V9$(),7,2),2)) TO C1$,N1$,B1$,A1
```

In this example, the next consecutive record is read. If the record is being held by another user, the system will wait 25 seconds before an error occurrs.

In line 60 the VAL function is used on the seventh and eigthth bytes of V9$() array to determine the number of bytes that were read.

CONSECUTIVE WRITE

General Form:

GOSUB '104 (file-id , efs , data)

Where:

file-id — is an alpha-numeric variable that represents the file identifier assigned to the file.

efs — is an alpha-numeric variable that represents the EFS header information for the file indicated by V9$ (the file identifier).

data — can be either an alpha-numeric literal or an array designator. If a literal string is used the information must be enclosed in double quotation marks.

PURPOSE:

The CONSECUTIVE WRITE ('104) subroutine enables you to write the next sequential record to a specified consecutive file.

To write the information contained in more than one variable to a file at one time, you can use the $PACK statement (or some other appropriate BASIC-2 statement) to pack the information into a single variable.

RETURNS:

The CONSECUTIVE WRITE subroutine returns the following information:

- V0$ is the return code. Refer to Appendix A for more information.
- V7$ is the EFS header information for the file indicated by V9$ (the file identifier). Refer to Appendix A for more information.
- V9$() contains the following five bytes of status information:
  - Byte 01 contains internal status information. This value should always be equal to 01.
  - Bytes 02 and 03 contain the number of bytes written.
  - Bytes 04 and 05 contain the number of bytes in the byte block and should be HEX(0000).

EXAMPLE: (CONSECUTIVE WRITE)

```
10 DIM C1$5, N1$30, B1$3, W$8, D$43, H$2, H1$32
20 H$=V9$:H1$=V7$
30 W$=HEX(A005A01EA0035205)
40 $PACK (F=W$) D$ FROM C1$, N1$, B1$, A1
50 GOSUB '104 (H$, H1$, D$)
60 IF V0$=HEX(FF) THEN 80
70 STOP "ERROR IN WRITE"
80 ...
```

In this example, the data held in the variable D$ is written to the file specified by H$.

CONSECUTIVE REWRITE

General Form:

GOSUB '105 (file-id , efs , len , data)

Where:

file-id	is an alpha-numeric variable that represents the file identifier assigned to the file.
efs	is an alpha-numeric variable that represents the EFS header information for the file indicated by V9$ (the file identifier).
len	indicates the length of the record to rewritten. The value of the len parameter must matcha the record length exactly, including trailing spaces.
data	can be either an alpha-numeric literal, or an array designator. If a literal string is used the information must be enclosed in double quotation marks.

PURPOSE:

The CONSECUTIVE REWRITE ('105) subroutine enables you to overwrite an existing record in a consecutive file. The record must have been previously read with the HOLD option.

To write the information contained in more than one variable to a file at one time using the REWRITE subroutine, you can use the $PACK statement to pack the information into a single variable or some other appropriate BASIC-2 statement.

RETURNS:

The CONSECUTIVE REWRITE subroutine returns the following information:

- V0$ is the return code. Refer to Appendix A for more information.
- V7$ is the EFS header information for the file indicated by V9$ (the file identifier). Refer to Appendix A for more information.

- V9$() contains the following three bytes of status information:
  - Byte 01 contains the number of words in the word block which should be equal to HEX(00).
  - Bytes 02 and 03 contain the number of bytes in the byte block which should be equal to HEX(0000).

EXAMPLE: (CONSECUTIVE REWRITE)

The following is an example of how to use the CONSECUTIVE REWRITE subroutine:

```
10 DIM C1$5, N1$30, B1$3, W$8, D$43, H$2, H1$32
20 W$=HEX(A005A01EA0035205)
30 $PACK (F=W$) D$ FROM C1$, N1$, B1$, A1$
40 GOSUB '105 (H$, H1$, 43, D$)
50 IF V0$ = HEX(FF) THEN 70
60 STOP "ERROR IN REWRITE"
70 ...
```

CONSECUTIVE SKIP

General Form:

```
GOSUB '106 (file-id , efs , nnnnnnnn)
```

Where:

file-id — is an alpha-numeric variable that represents the file identifier assigned to the file.

efs — is an alpha-numeric variable that represents the EFS header information for the specified file.

nnnnnnnn — is the number of records to skip; nnnnnnnn must be between 2^{+32} and -2^{+32}.

PURPOSE:

The CONSECUTIVE SKIP ('106) subroutine positions a consecutive file forward or backward a given number of records in the file. For example, if the first record of a file has been read, a SKIP value of 2 causes the next record read to be record 4. A SKIP value of -1 causes the same record to be reread by the next CONSECUTIVE READ ('103). A SKIP value of 0 is ignored.

RETURNS:

The CONSECUTIVE SKIP subroutine returns the following information:

- V0$ is the return code. Refer to Appendix A for more information.
- V7$ is the EFS header information for the file indicated by V9$ (the file identifier). Refer to Appendix A for more information.
- V9$() contains the following three bytes of status information:
  - Byte 01 contains the number of words in the word block which should be equal to HEX(00).
  - Bytes 02 and 03 contain the number of bytes in the byte block which should be equal to HEX(0000).

EXAMPLE: (CONSECUTIVE SKIP)

```
10 GOSUB '106 (H$, 30)
20 IF V0$=HEX(FF) THEN 50:STOP "ERROR IN SKIP"
```

In this example, the next 30 records in the specified file are skipped.

CONSECUTIVE LOCK

General Form:

GOSUB '107 (file-id , efs , mode)

Where:

file-id is an alpha-numeric variable that represents the file identifier assigned to the file.

efs is an alpha-numeric variable that represents the EFS header information for the specified file.

mode is the mode the file is opened in. The following lists the valid mode parameters and their meaning:

- R indicates the file is opened for read only access.
- S indicates the file is opened for shared access.
- X indicates the file is opened for exclusive access.
- E indicates the file is opened for extended access.

PURPOSE:

The CONSECUTIVE LOCK ('107) subroutine enables you to have exclusive rights to a consecutive file. No other program can access the file until you unlock the file.

RETURNS:

The CONSECUTIVE LOCK subroutine returns the following information:

- V0$ is the return code. Refer to Appendix A for more information.

- V7$ is the EFS header information for the file indicated by V9$ (the file identifier). Refer to Appendix A for more information.

- V9$() contains the following three bytes of status information:

  - Byte 01 contains the number of words in the word block which should be equal to HEX(00).

  - Bytes 02 and 03 contain the number of bytes in the byte block which should be equal to HEX(0000).

EXAMPLE: (CONSECUTIVE LOCK)

The following is an example of how to use the CONSECUTIVE LOCK subroutine:

```
  10 H$=V9$
  20 GOSUB '107 (H$, H1$, "X")
  30 IF V0$=HEX(FF) THEN 50
  40 STOP "ERROR IN LOCK"
  50 REM File was locked successfully.  Continue processing.
```

In this example, the "X" indicates the specified file is held for exclusive use. H1$ identifies the EFS header information.

CONSECUTIVE UNLOCK

General Form:

```
GOSUB '108 (file-id , efs)
```

Where:

file-id is an alpha-numeric variable that represents the file identifier assigned to the file.

efs is an alpha-numeric variable that represents the EFS header information for the specified file.

PURPOSE:

The CONSECUTIVE UNLOCK ('108) subroutine enables you to release a file from exclusive use, so that other programs can access the file.

RETURNS:

The CONSECUTIVE UNLOCK subroutine returns the following information:

- V0$ is the return code. Refer to Appendix A for more information.
- V7$ is the EFS header information for the file indicated by V9$ (the file identifier). Refer to Appendix A for more information.
- V9$() contains the following three bytes of status information:
  - Byte 01 contains the number of words in the word block which should be equal to HEX(00).
  - Bytes 02 and 03 contain the number of bytes in the byte block which should be equal to HEX(0000).

EXAMPLE: (CONSECUTIVE UNLOCK)

```
10 H$=V9$
20 GOSUB '108 (H$, H1$)
30 IF V0$=HEX(FF) THEN 50
40 STOP "ERROR IN UNLOCK"
50 REM Unlock was successful.  Continue processing.
```

In this example, specified file is released from exclusive use.

INDEXED READ

General Form:

```
GOSUB '109 (file-id , efs , hold , time , key , length , value)
```

Where:

file-id — is an alpha-numeric variable that represents the file identifier assigned to the file.

efs — is an alpha-numeric variable that represents the EFS header information for the specified file.

hold — indicates the hold option and can be either "H" or " ". "H" holds the record for exclusive processing. " " allows other programs to access the record.

time — is the amount of time in seconds the system will wait if the record is being held by another user. Specifying a value of zero (0) indicates that the system will wait indefinantly. Non-zero values should only be used when the file is opened in shared mode.

key — is the key path, either the primary (0) or alternate key (1-16).

length — is a numeric variable or expression that specifies the length of the key.

value — is an alpha-numeric variable or literal that indicates the value of the key.

PURPOSE:

The INDEXED READ ('109) subroutine enables you to read an indexed file. The file must have previously been opened. You can use either a primary or alternate key; for more information, see the General Notes section.

RETURNS:

The INDEXED READ subroutine returns the following information:

- V0$ is the return code. Refer to Appendix A for more information.
- V1 contains the length of data read (in bytes) in V9$() array.
- V4$ is the alternate key mask.
- V9$() the data read from the file.

GENERAL NOTES

DMS allows a primary key and up to 16 alternate keys. In the INDEXED READ subroutine, the primary key is indicated by a 0, the alternate keys are indicated by the numbers 1 through 16. An example of each is provided in the Example section.

The V9$() array is originally dimensioned to hold 4096 bytes of information (including the status information). However, you can decrease the size of the array depending on your needs.

EXAMPLE: (INDEXED READ)

```
10 DIM C1$5, N1$30, B1$3, W$8
20 W$=HEX(A005A01EA0035205)
30 C1$="00001"
40 GOSUB '109 (H$, H1$, "H", 0, 0, 5, C1$)
50 IF V0$=HEX(FF) THEN 70
60 STOP "ERROR IN INDEXED READ"
70 REM Successful indexed read.  Continue processing
80 $UNPACK (F=W$) STR(V9$(),1,V1) to K$,N1$,B1$,A1
```

In this example, the path is identified as the primary key by the zero (0) in the key parameter position. The key length is identified as 5 characters in length, and the value of the key is equal to the value of C1$. Note also that the hold option is used.

The following example shows how to indicate an alternate key.

```
10 DIM C1$5, N1$30, B1$3, W$8
20 W$=HEX(A005A01EA0035205)
30 C1$="00001"
40 GOSUB '109 (H$, H1$, "H", 0, 4, 5, C1$)
50 IF V0$=HEX(FF) THEN 70
60 STOP "ERROR IN INDEXED READ"
70 REM Successful indexed read.  Continue processing
80 $UNPACK (F=W$) STR(V9$(),1,V1) to K$,N1$,B1$,A1
```

In this example, the path is identified as the fourth alternate key by the four (4) in the key parameter position. The other parameter values remain the same.

INDEXED READ NEXT

General Form:

GOSUB '110 (file-id , efs , hold , time)

Where:

file-id	is an alpha-numeric variable that represents the file identifier assigned to the file.
efs	is an alpha-numeric variable that represents the EFS header information for the specified file.
hold	indicates the hold option and can be either "H" or " ". "H" holds the record for exclusive processing. " " allows other programs to access the record.
time	is the amount of time in seconds the system will wait if the record is being held by another user. Non-zero values should only be used when the file is opened in shared mode.

PURPOSE:

The INDEXED READ NEXT ('110) subroutine enables you to read an indexed file sequentially. The file must have previously been opened.

RETURNS:

The INDEXED READ NEXT subroutine returns the following information:

- V0$ is the return code. Refer to Appendix A for more information.
- V1 contains the length of data read (in bytes) in V9$() array.
- V4$ is the alternate key mask.
- V9$() the data read from the file.

EXAMPLE: (INDEXED READ NEXT)

```
10 DIM C1$5,N1$30,B1$3,W$8,K1$1,K2$5
20 W$=HEX(A005A01EA0035205)
30 GOSUB '110 (H$, H1$, "H", 25)
40 IF V0$=HEX(FF) THEN 60
50 STOP "ERROR IN READ NEXT"
60 REM Successful READ NEXT.  Continue processing
70 ...
```

In this example, the next record in the file indicated by H$ is read. If the record is being held by another user, the program waits 25 seconds before generating an error return code.

INDEXED WRITE

General Form:

GOSUB '111 (file-id , efs , alt , data)

Where:

file-id — is an alpha-numeric variable that represents the file identifier assigned to the file.

efs — is an alpha-numeric variable that represents the EFS header information for the specified file.

alt — represents the alternate key mask.

data — can be either an alpha-numeric, an array designator, or a literal. If a literal string is used the information must be enclosed in double quotation marks.

PURPOSE:

The INDEXED WRITE ('111) subroutine enables you to write a keyed record to an indexed file.

To write the information contained in more than one variable to a file at one time using the WRITE subroutine, you can use the $PACK statement (or some other appropriate BASIC-2 statement) to pack the information into a single variable.

RETURNS:

The INDEXED WRITE subroutine returns the following information:

- V0$ is the return code. Refer to Appendix A for more information.

EXAMPLE: (INDEXED WRITE)

```
10 DIM N1$30,B1$3,W$8,K1$1,K$5
20 W$=HEX(A005A01EA0035205)
30 PACK ( F=W$ ) D$ FROM C1$, N1$, B1$, A1$
40 GOSUB '111 (H$, H1$, H2$, D$)
40 IF V0$=HEX(FF) THEN 60
50 STOP "ERROR IN INDEXED READ"
60 REM Good indexed read.  Continue processing.
```

INDEXED REWRITE

General Form:

GOSUB '112 (file-id , efs , alt , len , data)

Where:

file-id — is an alpha-numeric variable that represents the file identifier assigned to the file.

efs — is an alpha-numeric variable that represents the EFS header information for the specified file.

alt — represents the alternate key mask.

len — indicates the length of the record to rewritten. The value of the len parameter must matcha the record length exactly, including trailing spaces.

data — can be either an alpha-numeric variable, an array designator, or a literal. If a literal string is used the information must be enclosed in double quotation marks.

PURPOSE:

The INDEXED REWRITE ('112) subroutine enables you to overwrite an existing record in an indexed file. The rewritten record size is the same as that of the existing record.

To write the information contained in more than one variable to a file at one time using the REWRITE subroutine, you can use the $PACK statement (or some other appropriate BASIC-2 statement) to pack the information into a single variable.

RETURNS:

The INDEXED REWRITE subroutine returns the following information:

- V0$ is the return code. Refer to Appendix A for more information.

GENERAL NOTES:

A record can be rewritten only if the record is read with the hold option equal "H".

EXAMPLE: (INDEXED REWRITE)

```
10 DIM C1$5,N1$30,B$3,W$8,D$43
20 W$=HEX(A005A01EA0035205)
30 $PACK ( F = W$ ) D$ FROM C1$, N1$, B1$, A1
40 GOSUB '112 (H$, H1$, H2$, 43, D$)
50 IF V0$=HEX(FF) THEN 70
60 STOP "ERROR IN INDEXED REWRITE"
70 ...
```

INDEXED DELETE

General Form:

GOSUB '113 (file-id , efs)

Where:

file-id	is an alpha-numeric variable that represents the file identifier assigned to the file.
efs	is an alpha-numeric variable that represents the EFS header information for the specified file.

PURPOSE:

The INDEXED DELETE ('113) subroutine enables you to delete a specific record from an indexed file.

RETURNS:

The INDEXED DELETE subroutine returns the following information:

- V0$ is the return code. Refer to Appendix A for more information.

GENERAL NOTES:

A record can be rewritten only if the record is read with the hold option equal "H".

EXAMPLE: (INDEXED DELETE)

```
  10 GOSUB '113 (H$, H1$)
  20 IF V0$ = HEX(FF) THEN 40
  30 STOP "ERROR IN INDEXED DELETE"
  40 REM Indexed delete successful.  Continue processing.
```

In this example, H$ indentifies the file that the INDEXED DELETE subroutine will operate on.

INDEXED FIND

General Form:

GOSUB '114 (file-id , efs , select , path , length , value)

Where:

file-id — is an alpha-numeric variable that represents the file identifier assigned to the file.

efs — is an alpha-numeric variable that represents the EFS header information for the specified file.

select — represents the selection critera. The following values are valid entries for this parameter:

- "8000" indicates equal to
- "2000" represent greater than
- "6000" indicates greater than or equal to

path — indicates either primary (0) or alternate key (1-16) path number.

length — is a variable or numeric expression that specifies the length of the key.

value — is a variable or an alpha or numeric expression that indicates the value of the key.

PURPOSE:

The INDEXED FIND ('114) subroutine enables you to read an indexed file based on a comparison expressed in the select input parameter to the primary or alternate key.

RETURNS:

The INDEXED FIND subroutine returns the following information:

- V0$ is the return code. Refer to Appendix A for more information.
- V1 contains the length of data read (in bytes) in V9$() array.
- V4$ is the alternate key mask.
- V9$() the data read from the file.

EXAMPLE: INDEXED FIND

```
   10 DIM H$2,H1$32,H3$2,K$5
   20 K$="00003":REM KEY VALUE
   30 K1=0:REM KEY PATH
   40 K2=5:REM KEY LENGTH
   50 H3$=HEX(2000):REM FIND CRITERIA = GREATERN
   60 REM FIND RECORD WITH A KEY VALUE GREATER THAN "00003"
   70 GOSUB '114(H$, H1$, H3$, K1, K2, K$)
   80 IF V0$=HEX(FF) THEN 90
   90 STOP "ERROR IN INDEXED FIND"
  100 REM GOOD INDEXED FIND.  CONTINUE PROCESSING."
  110 ...
```

In this example, the selection criterion is set to greater than (2000). This value indicates that the next record read will have a primary key value greater than the value of C1$.

INDEXED LOCK

General Form:

```
GOSUB '115 (file-id , efs , post , key , length , value)
```

Where:

file-id — is an alpha-numeric variable that represents the file identifier assigned to the file.

efs — is an alpha-numeric variable that represents the EFS header information for the specified file.

post — represents the position in the record that the key starts.

key — is the key path, either the primary (0) or alternate key (1-16).

length — is a numeric variable or literal that specifies the length of the key.

value — is alpha-numeric variable or an alpha-numeric literal that indicates the value of the key.

PURPOSE:

The INDEXED LOCK ('115) subroutine enables you to to have exclusive rights to an indexed file. No other program can access the file until you unlock the file.

You can use either the primary or alternate key. See the General Notes section for more information.

RETURNS:

The INDEXED LOCK subroutine returns the following information:

- V0$ is the return code. Refer to Appendix A for more information.

GENERAL NOTES

DMS allows a primary key and up to 16 alternate keys. In the INDEXED READ subroutine, the primary key is indicated by a 0, the alternate keys are indicated by the numbers 1 through 16..

EXAMPLE: (INDEXED LOCK)

```
10 H$ = V9$
20 GOSUB '115 (H$, H1$, P, K, L, K1$)
30 IF V0$=HEX(FF) THEN 50
40 STOP "ERROR IN LOCK"
50 REM Indexed file lock successful.  Continue processing.
60 ...
```

INDEXED UNLOCK

General Form:

GOSUB '116 (file-id , efs)

Where:

file-id	is an alpha-numeric variable that represents the file identifier assigned to the file.
efs	is an alpha-numeric variable that represents the EFS header information for the specified file.

PURPOSE:

The INDEXED UNLOCK ('115) subroutine enables you to release an indexed file so that other programs can access the file.

RETURNS:

The INDEXED UNLOCK subroutine returns the following information:

- V0$ is the return code. Refer to Appendix A for more information.

EXAMPLE: (INDEXED UNLOCK)

```
10 H$ = V9$
20 GOSUB '116 (H$, H1$)
30 IF V0$=HEX(FF) THEN 50
40 STOP "ERROR IN UNLOCK"
50 REM Indexed file unlock successful. Continue processing.
60 ...
```

RELATIVE READ

General Form:

```
GOSUB '117 (file-id , efs , hold , rec-num , number)
```

Where:

file-id — is an alpha-numeric variable that represents the file identifier assigned to the file.

efs — is an alpha-numeric variable that represents the EFS information for the specified file.

hold — indicates the hold option and can be either "H" or " ". "H" holds the record for exclusive processing. " " allows other programs to access the record.

rec-num — is the relative number of the record to be read.

number — is a numeric variable or expression indicating the number of relative records to be read.

PURPOSE:

The RELATIVE READ ('117) subroutine enables you to read a specified record in a relative file. The file must have previously been opened.

RETURNS:

The RELATIVE READ subroutine returns the following information:

- V0$ is the return code. Refer to Appendix A for more information.
- V7$ is the EFS header information for the file indicated by V9$ (the file identifier). Refer to Appendix A for more information.
- V9$() contains eight bytes of status information and the data read from the file. Refer to the RELATIVE READ General Notes section for more information.

GENERAL NOTES

In the RELATIVE READ subroutine, the V9$() array is used to store status information and the data read from the file. The first eight bytes of the array are used to store the status information. Table 4-2 describes the status information.

Table 4-2. RELATIVE READ Status Information

Byte(s)	Description
01	Contains the number of words in the word block. This value should always be equal to HEX(01).
02 and 03	Contain the number of records read.
04 and 05	Contain the number of bytes in the byte block.
06	Contains the datablock ID. This value should always be equal to 01.
07 and 08	Contains the number of bytes read. The value of these two bytes can be up to 64K.

The remaining bytes of the array (starting at the ninth byte) are used to store the data read from the file.If you know the length of the records read, you will know how many bytes of V9$ will be used to store the data. If you do not know the length of the records in the file, or the file contains variable length records you can use the VAL function on the 7th and 8th bytes of V9$() array to get the length of the data read. See the Example section for more information.

The V9$() array is originally dimensioned to hold 4096 bytes of information (including the status information). However, you can decrease the size of the array depending on your needs.

EXAMPLE: (RELATIVE READ)

The following is an example of how to use the RELATIVE READ subroutine:

```
  10 DIM C1$5,N1$30,B1$3,W$8,H$2,H1$32
  15 H$=V9$
  20 W$=HEX(A005A01EA0035205)
  30 GOSUB '117 (H$, H1$, "H", 100, 125, 1)
  40 IF V0$=HEX(FF) THEN 60
  50 STOP "ERROR IN RELATIVE READ"
  60 REM Good relative read.  Continue processing.
  70 $UNPACK (F=W$) Q$() to C1$,N1$,B1$,A1
```

In this example, the 125th record of the specified file is read. If the record is being held by another user, the system will wait for up to 100 seconds to read the record before an error occurrs.

RELATIVE WRITE

General Form:

```
GOSUB '118 (file-id , efs , number , data)
```

Where:

file-id is an alpha-numeric variable that represents the file identifier assigned to the file.

efs is an alpha-numeric variable that represents the EFS information for the specified file.

number is a numeric variable or expression indicating the number of relative records to be read.

data can be either an alpha-numeric variable, an array designator, or a literal. If a literal string is used the information must be enclosed in double quotation marks.

PURPOSE:

The RELATIVE WRITE ('118) subroutine enables you to write the next record to a specified relative file.

To write the information contained in more than one variable to a file at one time, you can use the $PACK statement (or some other appropriate BASIC-2 statement) to pack the information into a single variable.

RETURNS:

The RELATIVE WRITE subroutine returns the following information:

- V0$ is the return code. Refer to Appendix A for more information.
- V7$ is the EFS header information for the file indicated by V9$ (the file identifier). Refer to Appendix A for more information.
- V9$() contains the following five bytes of status information:
  - Byte 01 contains the number of words in the word block. This value should always be equal to HEX(01).
  - Bytes 02 and 03 contain the number of bytes written.
  - Bytes 04 and 05 contain the number of bytes in the byte block and should be HEX(0000).

EXAMPLE: (RELATIVE WRITE)

The following is an example of how to use the RELATIVE WRITE subroutine:

```
10 DIM C1$5,N1$30,B1$3,W$8,D$41,H$2,H1$32
20 W$=HEX(A005A01EA0035205)
30$PACK (F=W$) D$ FROM C1$,N1$,B1$,A1
40 GOSUB '118 (H$, H1$, 1, D$)
50 IF V0$=HEX(FF) THEN 70
60 STOP "ERROR IN RELATIVE WRITE"
70 REM Good relative write.  Continue processing.
80 ...
```

In this example, the value of D$ is written to the file as one record.

RELATIVE REWRITE

General Form:

GOSUB '119 (file-id , efs , number , len , data)

Where:

file-id — is an alpha-numeric variable that represents the file identifier assigned to the file.

efs — is an alpha-numeric variable that represents the EFS header information for the specified file.

number — represents the number of the relative record to be rewritten.

len — indicates the length of the record to rewritten. The value of the len parameter must matcha the record length exactly, including trailing spaces.

data — can be either an alpha-numeric variable, an array designator, or a literal. If a literal string is used the information must be enclosed in double quotation marks.

PURPOSE:

The RELATIVE REWRITE ('119) subroutine enables you to overwrite an existing record in a relative file. The record must have been previously read with the HOLD option.

To write the information contained in more than one variable to a file at one time using the REWRITE subroutine, you can use the $PACK statement to pack the information into a single variable or some other appropriate BASIC-2 statement.

RETURNS:

The RELATIVE REWRITE subroutine returns the following information:

- V0$ is the return code. Refer to Appendix A for more information.
- V7$ is the EFS header information for the file indicated by V9$ (the file identifier). Refer to Appendix A for more information.
- V9$() contains the following three bytes of status information:
  - Byte 01 contains the number of words in the word block and should be equal to HEX(00).
  - Bytes 02 and 03 contain the number of bytes written and should be equal to HEX(0000).

EXAMPLE: (RELATIVE REWRITE)

The following is an example of how to use the RELATIVE REWRITE subroutine:

```
10 DIM C1$5,N1$30,B1$3,W$8,D$43,H$2,H1$32
20 W$=HEX(A005A01EA0035205)
30 $PACK (F=W$) D$ FROM C1$,N1$,B1$,A1
40 GOSUB '104 (H$, H1$, 5, 43, D$)
50 IF V0$=HEX(FF) THEN 70
60 STOP "ERROR IN REWRITE"
70 REM Good relative rewrite. Continue processing.
80 ...
```

In this example, the 5th relative record of the specified file is rewritten to the file.

RELATIVE DELETE

General Form:

GOSUB '120 (file-id , efs , number)

Where:

file-id — is an alpha-numeric variable that represents the file identifier assigned to the file.

efs — is an alpha-numeric variable that represents the EFS header information for the specified file.

number — represents the number of the relative record to be deleted.

PURPOSE:

The RELATIVE DELETE ('120) subroutine enables you to delete a specific record from an relative file.

RETURNS:

The RELATIVE DELETE subroutine returns the following information:

- V0$ is the return code. Refer to Appendix A for more information.
- V7$ is the EFS header information for the file indicated by V9$ (the file identifier). Refer to Appendix A for more information.
- V9$() contains the following three bytes of status information:
  - Byte 01 contains the number of words in the word block and should be equal to HEX(00).
  - Bytes 02 and 03 contain the number of bytes written and should be equal to HEX(0000).

EXAMPLE: (RELATIVE DELETE)

```
  10 H$ = V9$:H1$ = V7$
  20 GOSUB '120 (H$, H1$, 5)
  30 IF V0$=HEX(FF) THEN 50
  40 STOP "ERROR IN DELETE"
  50 REM Good relative record delete.  Continue processing.
  60 ...
```

In this example, H$ indicates the file that the fifth relative record is deleted from.

BLOCK READ

General Form:

GOSUB '121 (file-id , efs , block-num)

Where:

file-id	is an alpha-numeric variable that represents the file identifier assigned to the file.
efs	is an alpha-numeric variable that represents the EFS header information for the specified file.
block-num	represents the block number of the the block to be read.

PURPOSE:

The BLOCK READ ('121) subroutine enables you to read a 2K block of information from a consecutive, indexed, or relative file.

RETURNS:

The BLOCK READ subroutine returns the following information:

- V0$ is the return code. Refer to Appendix A for more information.
- V9$() contains the data read from the file. This array is 2048 bytes long.

GENERAL NOTES:

Each block that is read contains 2048 bytes of data.

EXAMPLE: (BLOCK READ)

```
10 H$ = V9$:H1$ = V7$:B1=5
20 GOSUB '121 (H$, H1$, B1):REM Read block number 5.
30 IF V0$=HEX(FF) THEN 50
40 STOP "ERROR IN BLOCK READ"
50 REM Good block read.  Continue processing.
60 ...
```

In this example, block number five of the specified file is read.

BLOCK WRITE

General Form:

GOSUB '120 (file-id , efs , block-num , data)

Where:

file-id is an alpha-numeric variable that represents the file identifier assigned to the file.

efs is an alpha-numeric variable that represents the EFS header information for the specified file.

block-num represents the block number of the the block to be written.

data can be either an alpha-numeric literal or an array designator. If a literal string is used the information must be enclosed in double quotation marks.

PURPOSE:

The BLOCK WRITE ('122) subroutine enables you to write a 2K block of information to a consecutive, indexed, or relative file.

RETURNS:

The BLOCK WRITE subroutine returns the following information:

- V0$ is the return code. Refer to Appendix A for more information.

GENERAL NOTES:

Each block contains 2048 bytes of data. When you use the BLOCK WRITE subroutine, you want to write approximately 2048 bytes of data to the file.

EXAMPLE: (BLOCK WRITE)

```
  10 H$ = V9$:H1$ = V7$:B1=5
  20 GOSUB '122 (H$, H1$, B1, D$):REM Write block number 5.
  30 IF V0$=HEX(FF) THEN 50
  40 STOP "ERROR IN BLOCK WRITE"
  50 REM Good block write.  Continue processing.
  60 ...
```

In this example, block number five is written to the specified file.

FILE CREATE

General Form:

```
GOSUB '100 (file-name , org , mode , op-flag , create , alt-key)
```

Where:

file-name — represents the name of the file. The file name can include the //SYSTEM/VOLUME/LIBRARY/FILENAME. The file name can also be written as ///VOLUME/LIBRARY/FILENAME. The SYSTEM, LIBRARY, and FILENAME can be up to 8 characters in length. The VOLUME can be up to 6 characters in length.

org — is the organization of the file. The following lists the valid file organization parameters and their meaning:

- C indicates the file is a consecutive file.
- I indicates the file is an indexed file.
- R indicates the file is a relative file.
- B indicates the file is a block file.

mode — is the mode the file is opened in. The following lists the valid mode parameters and their meaning:

- R indicates the file is opened for read only access.
- S indicates the file is opened for shared access.
- X indicates the file is opened for exclusive access.
- E indicates the file is opened for extended access.

opt-flag — indicates whether the file is created or created and opened. "T" indicates the file is created. " " indicates the file is created and opened.

create — specifies the attribute data for the file. See the General Notes section for more information.

alt-key — specifies the alternate key information, if required. See the General Notes section for more information.

PURPOSE:

The FILE CREATE ('100) subroutine enables you to create a DMS file. The subroutine enables you to specify the file name, including the system, the library, and actual name of the file. The library, volume and file name are required; the system name is optional. The FILE CREATE subroutine also enables you to specify the file type (Indexed, Consecutive, or Relative,) and the access mode (Read Only, Shared, Exclusive, or Extended).

RETURNS:

The FILE CREATE subroutine returns the following information:

- V0$ is the return code. Refer to Appendix A for more information.
- V7$ is the EFS header information for the file indicated by V9$ (the file identifier). Refer to Appendix A for more information.
- V9$() contains the following three bytes of status information:
  - Byte 01 contains the number of words in the word block which should be equal to HEX(00).
  - Bytes 02 and 03 contain the number of bytes in the byte block which should be equal to HEX(0000).

GENERAL NOTES

You can use the VS CREATE utility to create and maintain data files. Through CREATE you can add, delete, modify or examine data in the data files created using the utility. For more information on the CREATE utility, refer to the VS File Management Utilities Reference.

The create input parameter requires 40 bytes of information for consecutive, indexed, and relative files. This parameter requires an additional 8 bytes of information if the file being created is an indexed file with more than one key. Appendix A explains the content of the information required for the create parameter.

EXAMPLE: FILE CREATE)

```
10 DIM N$32, T$1, M$1, A$62, A1$8
20 N$="//SYSTEM/ANYVOL/ANYLIB/FILENAME"
30 T$="I"
40 M$="S"
50 GOSUB '200 (N$, T$, M$, " ", A$, A1$)
60 IF V0$=HEX(FF) THEN 80
70 STOP "ERROR IN CREATE"
80 REM Good file create.  Continue processing.
90 ...
```

In this example variables are used to represent the name, organization, mode, file atributes and alternate key attribute data parameters. The opt-flag parameter value of " " indicates that the file is created and opened.

FILE DELETE

General Form:

GOSUB '201 (file-name)

Where:

file-name	represents the name of the file. The file name can include the SYSTEM/VOLUME/LIBRARY/FILENAME. The file name can also be written as ///VOLUME/LIBRARY/FILENAME. The SYSTEM, LIBRARY, and FILENAME can be up to 8 characters in length. The VOLUME can be up to 6 characters in length.

PURPOSE:

The FILE DELETE ('201) subroutine enables you to delete any DMS file.

RETURNS:

The FILE DELETE subroutine returns the following information:

- V0$ is the return code. Refer to Appendix A for more information.
- V1 is the length of valid data in V9$() array. What data ???
- V7$ is the EFS header information for the file indicated by V9$ (the file identifier). Refer to Appendix A for more information.
- V9$() contains three bytes of internal status information.

EXAMPLE: (FILE DELETE)

```
10 DIM N$32
20 N$="///ANYVOL/ANYLIB/FILENAME"
30 GOSUB '201 (N$)
40 IF V0$=HEX(FF) THEN 60
50 STOP "ERROR IN FILE DELETE"
60 REM File successfully deleted.  Continue processing.
70 ...
```

In this example, the file specified by N$ (///ANYVOL/ANYLIB/FILENAME) is deleted from the system.

FILE RENAME

General Form:

GOSUB '202 (old-name , new-name)

Where:

old-name represents the name of the file as it currently exists on the system. The file name can include the SYSTEM/VOLUME/LIBRARY/FILENAME. The file name can also be written as ///VOLUME/LIBRARY/FILENAME. The SYSTEM, LIBRARY, and FILENAME can be up to 8 characters in length. The VOLUME can be up to 6 characters in length.

new-name represents the name of the file as it will be known to the system after the subroutine executes. The file name can include the SYSTEM/VOLUME/LIBRARY/FILENAME. The file name can also be written as ///VOLUME/LIBRARY/FILENAME. The SYSTEM, LIBRARY, and FILENAME can be up to 8 characters in length. The VOLUME can be up to 6 characters in length.

PURPOSE:

The FILE RENAME ('202) subroutine enables you to rename any DMS file.

RETURNS:

The FILE RENAME subroutine returns the following information:

- V0$ is the return code. Refer to Appendix A for more information.
- V1 is the length of valid data in V9$() array. What data ???
- V7$ is the EFS header information for the file indicated by V9$ (the file identifier). Refer to Appendix A for more information.
- V9$() contains three bytes of internal status information.

<u>EXAMPLE</u>: (FILE RENAME)

```
  10 DIM O#32, N$32
  20 O$="///OLDVOL/OLDLIB/OLDNAME"
  30 N$="///NEWVOL/NEWLIB/NEWNAME"
  40 GOSUB '202 (O$, N$)
  50 IF V0$=HEX(FF) THEN 70
  60 STOP "ERROR IN FILE RENAME"
  80 REM File successfully renamed.  Continue processing.
  70 ...
```

In this example, the file specified by N$ (///ANYVOL/ANYLIB/FILENAME) is deleted from the system.

GET FILE ATTRIBUTES

General Form:

GOSUB '203 (file-name)

Where:

file-name represents the name of the file. The file name can include the SYSTEM/VOLUME/LIBRARY/FILENAME. The file name can also be written as ///VOLUME/LIBRARY/FILENAME. The SYSTEM, LIBRARY, and FILENAME can be up to 8 characters in length. The VOLUME can be up to 6 characters in length.

PURPOSE:

The GET FILE ATTRIBUTES ('203) subroutine enables you to retrieve the value of one or more attributes associated with the specified file. The file must be opened first.

RETURNS:

The GET FILE ATTRIBUTE subroutine returns the following information:

- V0$ is the return code. Refer to Appendix A for more information.
- V1 is the length of valid data in V9$() array. What data ???
- V7$ is the EFS header information for the file indicated by V9$ (the file identifier). Refer to Appendix A for more information.
- V9$() contains three bytes of internal status information. See Appendix A for an explanation of the attributes returned.

EXAMPLE: (GET FILE ATTRIBUTES)

```
  10 DIM N$32
  20 N$="///ANYVOL/ANYLIB/FILENAME"
  30 GOSUB '203
  40 IF V0$=HEX(FF) THEN 60
  50 STOP "ERROR IN GET FILE ATTRIBUTES"
  60 REM Good read on file attributes.  Continue processing."
  70 ...
```

In this example, the file attributes for the file ///ANYVOL/ANYLIB/FILENAME are returned and held in V9$() array.

APPENDIX-A
ADDITIONAL INFORMATION

A.1 INTRODUCTION

This appendix contains additional information on the following:

- Return codes
- The Extended File Sharing (EFS) information contained in variable V7$
- The file attribute information contained in variable V9$

A.2 RETURN CODES

The return codes generated by the DMS Access Subroutines are represented in HEX format. Table A-1 lists the return codes generated by the DMS Access Subroutines with their corresponding explanation.

Table A-1. Return Codes

Return Code Value	Explanation	Recovery Action
00	Indicates an error in the subroutine call.	Check your code for possible programming or syntax errors. Check the EFS header for more information.
5A	Indicates an error in the 2258 firmware.	Make sure the 2200/VS LCO task is operational and that the DMS task has been assigned. Check the EFS header for more information.
FF	No errors detected.	If an error is present, check the EFS header for more information.

A.3 EXTENDED FILE SHARING (EFS) HEADER

The DMS Access Subroutines uses the Extended File Sharing (EFS) protocol for controlling file information. The EFS information is 32 bytes long. The DMS Access Subroutines store the EFS header information in variable V7$.

Table A-2 describes the information contained in the EFS header.

Table A-2. EFS Information

Byte(s)	Explanation	Initial Value (in HEX)
01 through 4	Contain identification information.	FF534D42
05	Is required. If it is other HEX(FF), an error has occurred.	FF
06	Contains the error class. See the Error Classes section in this appendix for more information.	00
07	Contains the extended command code. This value is initialized depending on the subroutine used.	(Initialized by subroutine invoked.)
08 and 09	Contain a two byte error code. See the Error Codes section in this appendix for more information.	0000
10	Is reserved for future use.	00
11 through 22	Are reserved for future use.	All zeros.
23 and 24	Are required.	FFFF
25 and 26	Reserved for future use.	0000
27 and 28	Reserved for future use.	0000

(continued)

Table A-2. EFS Information (continued)

Byte(s)	Explanation	Initial Value (in HEX)
29 and 30	Contain the User Id number. This value is updated when the GENERAL OPEN subroutine is used.	0000
31 and 32	Reserved for future use.	0000

A.3.1 Error Classes

The error class indicates which task was invoked when the error occurred. Table A-3 explains the values returned for the error classes.

Table A-3. Error Classes

Value	Explanation
07	Is reserved for OMT.
08	Indicates the CATALOG server.
09	Indicates the FILE server.
10	Indicates the WITA server.
11	Indicates the PRINT server.
12	Indicates the QLI server.
13	Indicates the QLI:FORMATER server.
14	Indicates the QUEUE:JOB server.
15	Indicates the DMPACK server.
20	Indicates the User server.

A.3.2 Error Codes

The error code indicates why the error occurred. Table A-4 explains the values returned for the error codes.

Table A-4. Error Codes

Value	Explanation
0	Indicates no error was detected.
1	Indicates that an invalid function was specified to the server indicated by the error class.
2	Indicates that the file was not found.
3	Indicates that the library indicated was not found.
4	Indicates that too many files were opened.
5	Indicates that the user has insufficient access rights.
6	Indicates that an invalid file identifier was specified.
7	Indicates that there was a server processing error.
8	Indicates that there was insufficient space allocated to perform the required function.
9	Indicates that there was a VTOC error.
10	Indicates that invalid parameters were found for the function required.
11	Indicates an invalid file format was found.

(continued)

Table A-4. Error Codes (continued)

Value	Explanation
12	Indicates that an invalid open access mode was found.
13	Indicates that there was a disk space or a disk space extents error.
14	Indicates that an invalid function was found for the IO mode.
15	Indicates that the volume requested is not mounted.
16	Indicates that delete errors where found.
17	Indicates that an invalid device was found.
18	Indicates that a NODATA Read was attempted on a file opened in shared mode.
19	Indicates that an invalid function was found during a relative read.
20	Indicates that the file already exists.
21	Indicates a file possession conflict.
22	Indicates an invalid key size.
23	Indicates an invalid key value.
24	Indicates a Boundary violation occurred.
25	Indicates that the end of file has been found.
26	Indicates an invalid attempt to REWRITE a compressed record.

(continued)

Table A-4. Error Codes (continued)

Value	Explanation
28	Indicates an invalid alternate key was found.
29	Indicates that an invalid function was specified for an alternate indexed file.
30	Indicates a permanent I/O error was found.
31	Indicates that an undefined position was specified for the READ NEXT function.
32	Indicates disk problems were encountered.
33	Indicates that recovery problems were encountered.
34	Indicates that the file organization needs to be specified.
36	Indicates that an invalid WRITE was issued to a relative file.
37	Indicates an invalid function was specified for a file opened in shared mode.
38	Indicates an invalid START function for a file opened in no-shared mode.
39	Indicates an invalid START for PAM access method.
40	Indicates the requested device is in use.
41	Indicates the requested device is not attached.
42	Indicates that access was denied.

(continued)

Table A-4. Error Codes (continued)

Value	Explanation
43	Indicates an invalid sequence for delete, BAM access method, or for a REWRITE to a consecutive file was encountered.
44	Indicates that a START WAIT was issued but that no I/O was pending.
45	Indicates that no wait was issued for the previous I/O.
46	Indicates that there was a timeout on a shared mode resource wait.
47	Indicates that an indexed file was requested but that FDR indicates the file is not an indexed file.
48	Indicates an attempt to compress data in a relative file.
96	Indicates that severe DMS errors were encountered.
98	Indicates that task problems were encountered an that the task indicated by the error class should be restarted.
99	Indicates that a non-specific file system error was encountered.

A.4 FILE ATTRIBUTE INFORMATION

This section explains the file attribute information returned when you use the GENERAL OPEN ('101) or GET FILE ATTRIBUTE ('203) subroutines.

The file attribute data contains 62 bytes of information for consecutive, indexed, and relative files. This data contains an additional 8 bytes of information if you are working with an indexed file with more than one key. Table A-5 explains the initial 62 bytes of file attribute information.

Table A-5. File Attribute Data

Byte(s)	Description
01	Indicates the file organization: C for consecutive, I for indexed, R for relative, or B for block.
02	Indicates the record format: F for fixed length records or V for variable length records.
03	Indicate the compression flag: Y indicates compression is used, N indicates compression is not used.
04	Indicates the file class: A-Z or #, $, or @.
05 through 08	Indicate the approximate number of records the file contains.
09 through 12	Indicate the number of bytes in each record.
12 through 16	Indicate the number of extents of disk space allocated.
17 through 20	Indicate the number of blocks in the file.
21 through 24	Indicate the number of blocks allocated for the file.
25 through 32	Indicate the name of the person who created the file.
33 through 38	Indicate the date the creation date of the file. Format is YYMMDD.
39 through 44	Indicate the last date the file was modified. This date is the same as the creation date when creating the file. Format is YYMMDD.
45 through 50	Indicate the expiration date of the file. Format is YYMMDD.
51	Indicates if the file has a WP prologue and must be N for no.

(continued)

Table A-5. File Attribute Data (continued)

Byte(s)	Description
52 and 53	Indicate the primary key position for indexed files. For non-indexed files this value is HEX(0000).
54 and 55	Indicate the primary key length for indexed files. For non-indexed files this value is HEX(0000).
56 and 57	Indicate the number of alternate keys for indexed files. Indexed files can have between 1 and 16 alternate keys. For non-indexed files this value is HEX(0000).
58 and 59	Indicate the alternate key mask for indexed files. For non-indexed files this value is 0000
60 through 62	Indicate the file access. ??? for write, ??? for read, or ??? for execute.

Table A-6 explains the file attribute information returned or required for indexed files with alternate keys.

Table A-6. Alternate Key File Attribute Data

Byte(s)	Description
01	Indicates whether records with duplicate alternate keys are allowed. Y indicates that duplicates are allowed. N indicates duplicates are not allowed.
02	Indicates whether the alternate key is to be compressed. Y indicates the key is compressed. N indicates the key is not compressed.
03 and 04	Indicate the ordinal number.
05 and 06	Indicate the start position of the alternate key.
07 and 8	Indicate the length in bytes of the alternate key.

A.5 FILE CREATE INFORMATION

This section explains the information required for the create input parameter for the FILE CREATE ('100) subroutine.

The create paramete requuires 40 bytes of information for consecutive, indexed, and relative files. This parameter requires an additional 8 bytes of information if you are creating an indexed file with more than one key. Table A-7 explains the initial 40 bytes of information.

Table A-7. Create Parameter Data

Byte(s)	Description
01	Indicates the file organization: C for consecutive, I for indexed, R for relative, or B for block.
02	Indicates the record format: F for fixed length records or V for variable length records.
03	Indicates the file class: A-Z or #, $, or @.
04	Indicate the compression flag: Y indicates compression is used, N indicates compression is not used.
05 through 08	Indicate the approximate number of records the file contains.
09 through 12	Indicate the number of bytes in each record.
13 through 16	Indicate the number of blocks allocated for the file.
17 through 20	Indicate the size of blocks in the file.
21 through 26	Indicate the expiration date of the file. Format is YYMMDD.
27	Indicates if the file has a WP prologue and must be N for no.
28 and 29	Indicate the primary key position for indexed files. For non-indexed files this value is HEX(0000).
30 and 31	Indicate the primary key length for indexed files. For non-indexed files this value is HEX(0000).
32 and 33	Indicate the number of alternate keys for indexed files. Indexed files can have between 1 and 16 alternate keys. For non-indexed files this value is HEX(0000).
34	Indicates whether to take the created space and must be N for no.

(continued)

Table A-7. Create Parameter Data (continued)

Byte(s)	Description
35	For print files only; otherwise HEX(00). Indicates the print form number. Print form numbers are established by your site operations.
36	For print files only; otherwise HEX(00). Indicates the print class. Print classes are established by your site operations.
37	For print files only; otherwise HEX(00). Indicates the printer device number. Printer device numbers are established by your site operations.
38	For print files only; otherwise HEX(00). Indicates the number of copies to be printed.
39	For print files only; otherwise HEX(00). Indicates the status; either R for release or H for hold.
40	For print files only; otherwise HEX(00). Indicates the disposition; either R for release or H for hold.

Table A-8 explains the create data required for indexed files with alternate keys.

Table A-6. Create Data for Files With Alternate Keys

Byte(s)	Description
01	Indicates whether records with duplicate alternate keys are allowed. Y indicates that duplicates are allowed. N indicates duplicates are not allowed.
02	Indicates whether the alternate key is to be compressed. Y indicates the key is compressed. N indicates the key is not compressed.
03 and 04	Indicate the ordinal number, 0 through 15.
05 and 06	Indicate the start position of the alternate key. The start position is zeo based.
07 and 8	Indicate the length in bytes of the alternate key.