Access Manager™
Productivity Tool
Reference Manual

Digital Research
P.O. Box 579
160 Central Avenue
Pacific Grove, CA 93950
(408) 649-3896
TWX 910 360 5001

The Access Manager Reference Manual was prepared using the Digital Research TEX Text Formatter and printed in the United States of America.

****
* First Edition: March 1983 *
****

Foreword

Access Manager™ is a general purpose, keyed, file accessing package for 8080 or 8086 compatible microprocessors which run under CP/M® or MP/M™ operating systems. It is designed for programmers creating application packages that must access data records on the basis of identifying key values, such as a name or a number.

The Access Manager hardware environment must include an 8080, 8085, 8088, 8086, or Z80® microprocessor, at least 48K bytes of Random Access Memory (RAM), at least one random access secondary storage device such as a floppy or hard disk drive, and a console.

Access Manager is a set of routines called from programs written in CB80™ , CB86™ , PL/I-80™ , PL/I-86™ , Pascal/MT+™ , and Pascal/MT+86™ ; Access Manager is not a stand-alone data base management system. The routines require approximately 5K to 11K excluding buffer areas. In a multiple-user environment, Access Manager requires a separate memory segment of at least 20K bytes, a minimum of three pages (768 bytes) of common memory for special Access Manager queues, and 2K bytes of memory in each user's memory segment. Up to eight users can share Access Manager under MP/M if there is enough queue space.

Access Manager Documentation Set

This manual describes Access Manager, a programming productivity tool from Digital Research. It is important that you understand how this manual is organized and what it contains for it to serve you well.

There are three manuals in this documentation set. The first is the Access Manager Reference Manual. Contained here are descriptions of the product, instructions for using it, and numerous examples of how it can be applied in your application programs.

The second manual is the Access Manager Programmer's Guide for either the 8080 or 8086 implementations. Your set contains one of the following:

- Access Manager Programmer's Guide for the CP/M Family of Operating Systems.
- Access Manager Programmer's Guide for the CP/M-86 Family of Operating Systems.

Here you will find information for using Access Manager with a specific operating system and programming language. Your Programmer's Guide also contains facts concerning basic requirements and design constraints for Access Manager. Finally, there are

several examples showing how to organize your application program for the various languages that are supported.

The third manual in your documentation set is the Access Manager Function Summary. The summary is a compact, abbreviated version of information you need when using Access Manager. As you become familiar with the Access Manager functions, you will rely less on your Reference Manual and Programmer's Guide and more on the Function Summary.

Prerequisites

These manuals assume you are knowledgeable about keyed file accessing methods. If you are not familiar with these methods, the following reading is recommended:

Knuth, D.E. The Art of Computer Programming. Vol. 3, Sorting and Searching. Reading, Massachusetts: Addison-Wesley Publishing Company, 1973.

Wiederhold, G. Database Design. New York, New York: McGraw-Hill Book Company, 1977.

How to Use this Documentation

Your Access Manager documentation contains several forms of indexing to help you find information quickly and efficiently.

- Your Reference Manual and Programmer's Guide each contain a Table of Contents and subject index.
- Appendix A of your Reference Manual contains a function index. All the functions you can perform with Access Manager are listed alphabetically and cross-referenced to the mnemonic function name.
- The function descriptions in Section 3 are arranged alphabetically by their mnemonic function names.

Take a few moments to understand how the indexing facilities are constructed in this manual, and you will be able to find the information you need when you need it.

Conventions Used in this Documentation

Access Manager is designed for use in two distinct environments: single-user and multiple-user. A single-user environment consists of a single microcomputer, a single terminal, and one person to operate them. A multiple-user environment (which we hereafter shorten to multiuser), consists of a single microcomputer, two or more terminals, and two or more people simultaneously running programs on the computer.

With a few exceptions, the rules for using Access Manager are the same in single-user and multiuser environments. When a distinction is necessary in the procedure, the text is preceded by [SINGLE] or [MULTI], whichever is applicable.

Hexadecimal values are noted by the letter H appended to the number. For example, 20H represents a hexadecimal value of 20; 0FFH represents hexadecimal FF.

All examples shown in your Access Manager Reference Manual are coded in CBASIC® Compiler language. Similarities to PL/I and Pascal/MT+ should be apparent as the logic is the same in all cases.

Table of Contents

Table of Contents
(continued)

Table of Contents
(continued)

Tables, Figures, and Listings

Tables

Tables, Figures, and Listings (continued)

Figures

Listings

Section 1
Product Description

1.1 What is Access Manager?

Think about your public library for a moment. There is a staggering volume of information located there. Yet, the card indexes make it possible to find specific information quickly and easily. Access Manager is something like the card index files in the public library. But, Access Manager provides a way to find specific information on computer disks rather than between the covers of a book.

The librarian prepares the card index in the public library based on specific subject matter, with Access Manager you decide how to index the information in your computer files.

Access Manager is a general purpose, keyed, file accessing method for microcomputers. It is designed for use with application programs that need to access data records based on identifying key values, such as a name or a number.

1.2 Access Manager Architecture

Access Manager is comprised of several different functions that you can call from your application program. These functions are categorized according to basic purpose. This categorization has no effect on how you use Access Manager, but does help explain its logical structure.

Access Manager functions are categorized into the following groups:

- System Initialization and Maintenance
- Index File Setup and Maintenance
- Index File Updates
- Index File Search
- Data File Setup and Maintenance
- Data File Updates
- Data File and Record Locking

1.2.1 System Initialization and Maintenance

These are commonly known as housekeeping functions. They prepare Access Manager to work with your application program by indicating how you want to handle errors, how many data files will be used by the program, and more. Normally, your program will only use these functions at the beginning.

1.2.2 Index File Setup and Maintenance

These functions ready your index files for use by Access Manager and your application program. They are used to open a file, close it, or save updates made to it. There are also functions for determining the size of the file and erasing it when desired.

1.2.3 Index File Updates

Update functions are used to add information to or remove it from an index file. You can also change the information the index uses to point to its associated data records. These pointers are called data record numbers.

1.2.4 Index File Searches

To locate a data record in a data file, you must first find its corresponding entry in the matching index file. Index File Search functions help you find the index file entry. They can be used to find specific entries in the index or entries relative to some other value. For example, with these functions you can find the first entry in an index, the last entry, a specific entry, an entry that is greater or less than one you specify, and more.

1.2.5 Data File Setup and Maintenance

These functions ready a data file for use by Access Manager and your application program. For example, they can be used to open a data file, close it, or save updates made to it at critical points in your program. There are also functions to aid in counting the entries (or records) in a data file or to erase it when necessary.

1.2.6 Data File Updates

You can use these functions to make actual changes in a data file. For example, there is a function to read a record from the file and another to write a record into it.

1.2.7 Data File and Record Locking

In multiuser environments, it is often necessary to protect information in a data file to prevent its use at critical times; such as while a customer's balance is being updated. There are Access Manager functions to prevent anyone from using a data file or data record at these critical points.

The wide variety of Access Manager functions provides you with simple methods to accomplish the following:

- create and/or maintain data files.
- create and/or maintain indexes for your data files.
- retrieve information from your data files.
- protect the integrity of your data files when they are being used simultaneously by two or more people.

1.3 Access Manager Benefits

There are numerous benefits to using Access Manager. Here are just a few:

- simplified programming
- fast data retrieval
- language portability

1.3.1 Simplified Programming

Standardizing access methods for keyed data files across single-user and multiuser environments can be a complex programming task. This is further complicated by the variety of programming languages in use. Access Manager overcomes these complications by being readily compatible with various operating system environments and most program languages.

Furthermore, you will find application programs can be developed faster and are much less prone to error when Access Manager is used.

1.3.2 Fast Data Retrieval

Most keyed access methods for retrieving information from a data file are slow compared to the computer's processing speed. Consequently, many otherwise efficient application programs slow to a snail's pace when retrieving information under these conditions. Access Manager overcomes this problem with its unique indexing structure. The structure (known as a B-Tree and described later in this section) minimizes the number of times an index file has to be read to locate a data record. Under typical circumstances, Access Manager can locate a specific record in a file of one-half million records with a maximum of four disk accesses. This is not only fast, it is efficient use of your hardware.

With Access Manager you can reduce disk accesses by properly allocating disk buffers. Each buffer holds an index file record and is shared by all the index files. A least-recently-used priority scheme manages the assignment of index records to buffers. Of course, Access Manager checks to see if the required index record is in a buffer before needlessly accessing the disk.

1.3.3 Language Portability

Access Manager can be used with a variety of programming languages. And because the functions are standardized, application programs written in different languages are able to access a common data base. For example, in a multiuser environment, one user can be accessing the data base via an application program written in PL/I-80, another user accesses the same data base through a Pascal/MT+ program, and yet another via a CBASIC program.

1.4 Concepts and Facilities

Access Manager uses the following concepts and facilities to efficiently create, index, share, and recreate your index and data files:

- file structures
- B-Tree structures
- data locking
- file recreation

1.4.1 File Structures

Access Manager provides the necessary functions to create and manage index files and data files on secondary storage devices, such as floppy or hard disks.

Data Files

Every record in a data file used with Access Manager must be the same length. That is, every record must contain the same number of bytes, and the record length must be at least four bytes.

Data Records

Unlike data base systems, Access Manager treats a data record as a single entity, not a collection of fields. This means your application program must parse each data record into the required fields when necessary. One or more fields in a data record can be used as the key to that particular record. Normally, the key is a name, number, or other value uniquely identifying the contents of the data record.

Index Files

Index files contain the key value for each record and its assigned data record number. With just a key value, the index files make it possible for an application program to locate the associated data record (even allowing for duplicate keys) without a lengthy search of the data file. Access Manager creates index files using a height-balanced, multiway tree structure known as a B-Tree. This index structure guarantees the least number of disk accesses to search an index file. Besides being speedy, B-Trees eliminate the need to reorganize the index files.

Access Manager assumes nothing about the relationships between index and data files. Your application program must interpret the contents of the index files and how, if at all, they relate to the data files. Because Access Manager does not explicitly link index and data files, you can create key values in any way that suits your application.

Access Manager's separate index and data files permit flexibility in your program design. For example, several index files can reference one data file, or an index file can be referenced without a companion data file. For exceptionally large data files, one index file can reference many volumes. However, this flexibility leaves you with the responsibility for maintaining consistency between your index and data files.

Index and data file concepts used by Access Manager are discussed in detail under the ADDKEY function description in Section 3.

1.4.2 B-Tree Index Structure

A B-Tree is a height-balanced, multiway tree structure. Under this structure, the tree is inverted with the root at the top and the nodes at the bottom. Access Manager uses a variant of the B-Tree structure known as a B+ Tree where all key values are stored at the bottommost level of the tree.

Figure 1-1 shows part of a simple B+ Tree structure. To locate the data record with a key value of 180, the following steps are taken:

1) Access the root node.

2) Because 180 lies between 150 and 270, the middle branch is taken to access node 12.

3) 180 is greater than 178, so the right branch is taken to access node 4.

4) Because a match is found in node 4, the data record corresponding to key value 180 is record number 9 in the data file.

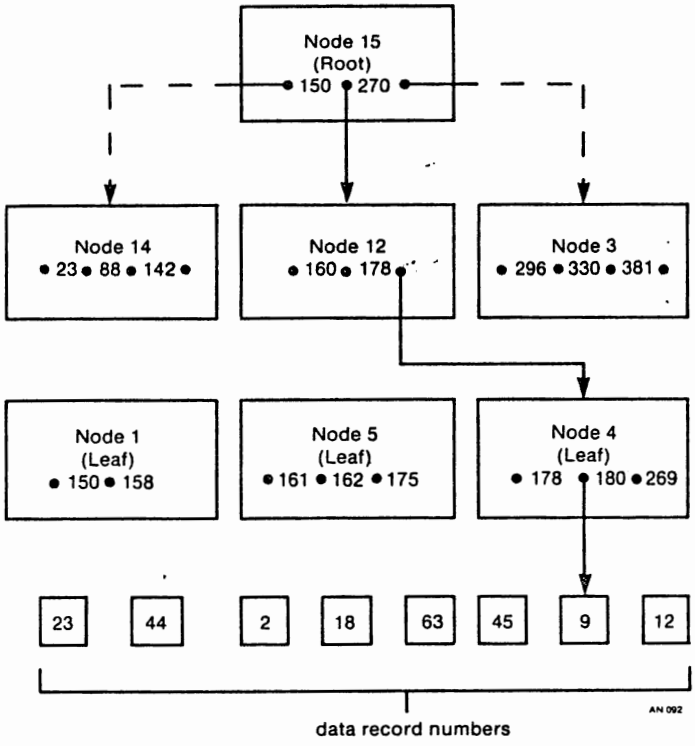

Figure 1-1. B+ Tree Structural Diagram

Consider these key points regarding Access Manager's index structures:

- In a multiway tree structure, many branches can emanate from each node of the tree. By comparison, in a binary tree there are at most two branches from each node. The advantage of additional branches is the height of the tree decreases as the number of branches per node increases; and the height of the tree determines the maximum number of disk accesses to search the tree.

For example, if Access Manager is set up with 512-byte nodes, and the length of the key values is ten bytes, up to 34 key values can be stored in each node. This ensures that a tree structure with no more than four levels can store 192,780 key values under worst case conditions. By contrast, the same number of key values in a binary tree would require 18 levels.

- A height-balanced tree structure ensures that all nodes at the bottom of the tree are equadistant from the root node. Therefore, no key value ever has a long access path. A height-balanced tree eliminates the problems of overflow areas with unpredictable access times.

- Placing all key values in the leaf nodes speeds and simplifies sequential key value accesses because Access Manager links the nodes in both ascending and descending key value order.

Any time you add a key to or delete one from an index file using Access Manager, the required changes in the B+ Tree structure are made automatically. This guarantees that key searches always remain efficient. Reorganization of the index structure is not necessary even after thousands of updates to the index file.

If you want further information on B-Tree structures, the following reading material is recommended:

Comer, D. "The Ubiquitous B-Tree." ACM Computing Surveys 11, no. 2(June 1979): 121-137.

Knuth, D.E. The Art of Computer Programming. Vol. 3, Sorting and Searching, 451-479. Reading, Massachusetts: Addison-Wesley Publishing Company, 1973.

1.4.3 Data Locking

To ensure that data files maintain their integrity in multiuser environments, Access Manager provides the ability to lock data records and/or data files. The locking facilities make it possible for the same or different application programs running concurrently to share and/or update index and data files at the same time.

The locking facilities take two forms: shared locks and exclusive locks. Shared locks permit separate users simultaneous access to the same data file or data record. An exclusive lock can only be held by one user at a time and bars all other users from having access to the locked data file or data record.

Data Record Locks

An application program can request a shared or exclusive record lock on individual data records. Any number of users can hold shared record locks on the same data record at the same time. However, only one user can hold an exclusive record lock on a particular data record. Used correctly, the locks allow several users to access a data record, but only one to update it. Access Manager record locks are set at the logical record level, not the physical or logical sectors of the operating system or storage medium.

Data File Locks

Besides locking data records, Access Manager can set locks on individual data files. Once a user holds an exclusive data file lock, all other requests for file or record locks on that file are refused. A shared data file lock signals your intent to use and possibly update a data file without the need to block other users from it.

Note: the locking facilities pertain only to data files and data records. The only way to exclude other users from accessing an index file is by using passwords.

1.4.4 File Recreation

Access Manager contains a utility program for quickly recreating index and data files when file integrity has been lost. You can set up a special file containing parameters which tell the utility program precisely how to reconstruct the files. Thus, file reconstruction becomes a simple task when necessary. Section 5 contains a complete description of the RECREATE utility program.

1.5 Application Program Structure

This subsection discusses common structures for application programs that use Access Manager. However, you should not conclude from this discussion that other program structures or uses of Access Manager are inappropriate.

You will find the suggested program structures in Figures 1-2 and 1-3 more meaningful after you have studied the individual function descriptions in Section 3.

Figure 1-2 outlines the program flow in a single-user environment. If your application is intended for a multiuser environment, the program flow in Figure 1-3 is more appropriate. Remember, an application program that runs in a multiuser environment also runs, without modification, in a single-user environment. For this reason, you might want to consider designing all of your application programs for a multiuser environment.

If an application program requires more than 65,535 entries in an index file or records in a data file, you will want to modify the program structures outlined in Figures 1-2 and 1-3 to call the DATVAL function after any other Access Manager function that returns a data record number. Call the SETDAT function before any other Access Manager function that accepts a data record number as an input parameter.

To simplify the program structures, error handling routines have been omitted from Figures 1-2 and 1-3. If you do not want error trapping, no changes are required to the program structures shown. This structure causes user error messages to display or print on the console and then return control to your operating system. If you want to control program actions when user errors occur, call the INTUSR function with a nonzero ERROPT% parameter. In this case, you should call the ERRCOD function after each call to the other Access Manager functions. If ERRCOD returns with a nonzero value, you can either branch to your error handling routines to recover, or perform an orderly shutdown of the application.

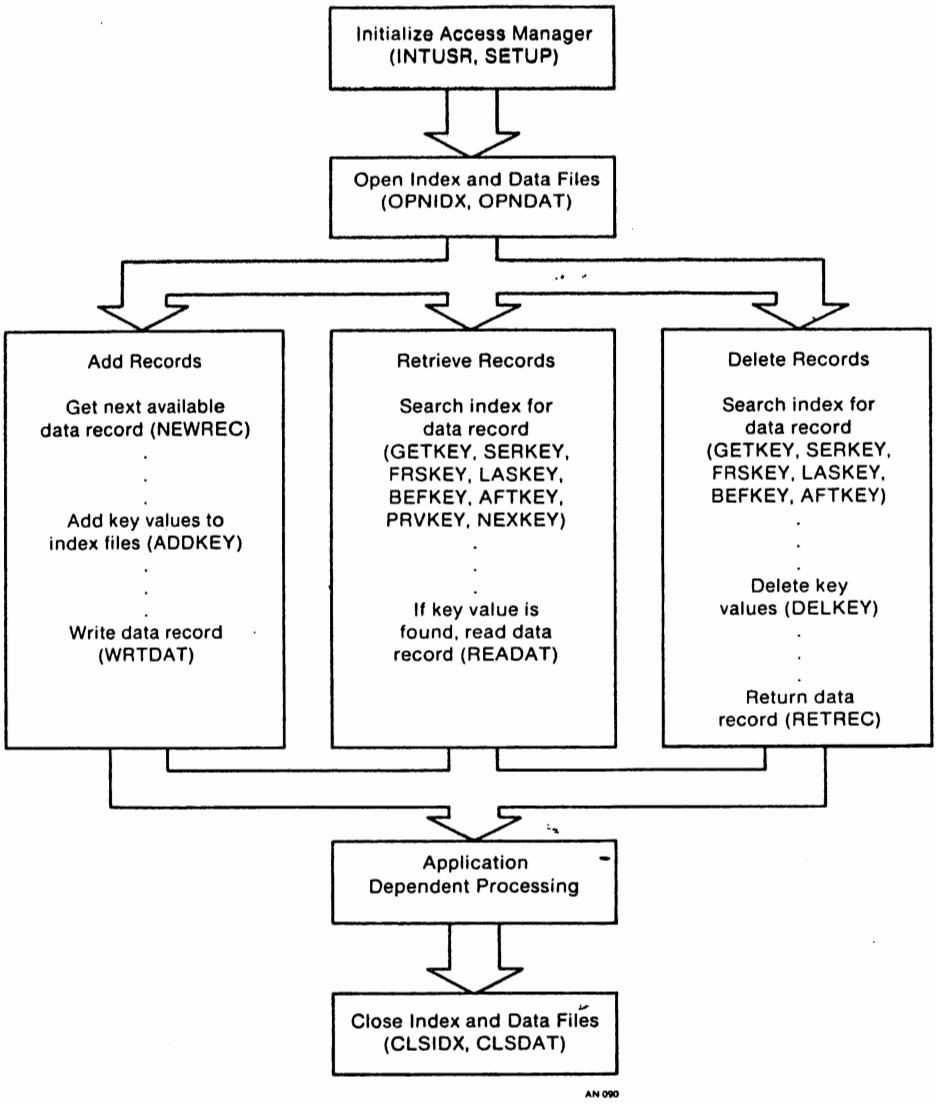

Figure 1-2. Single-user Program Flow

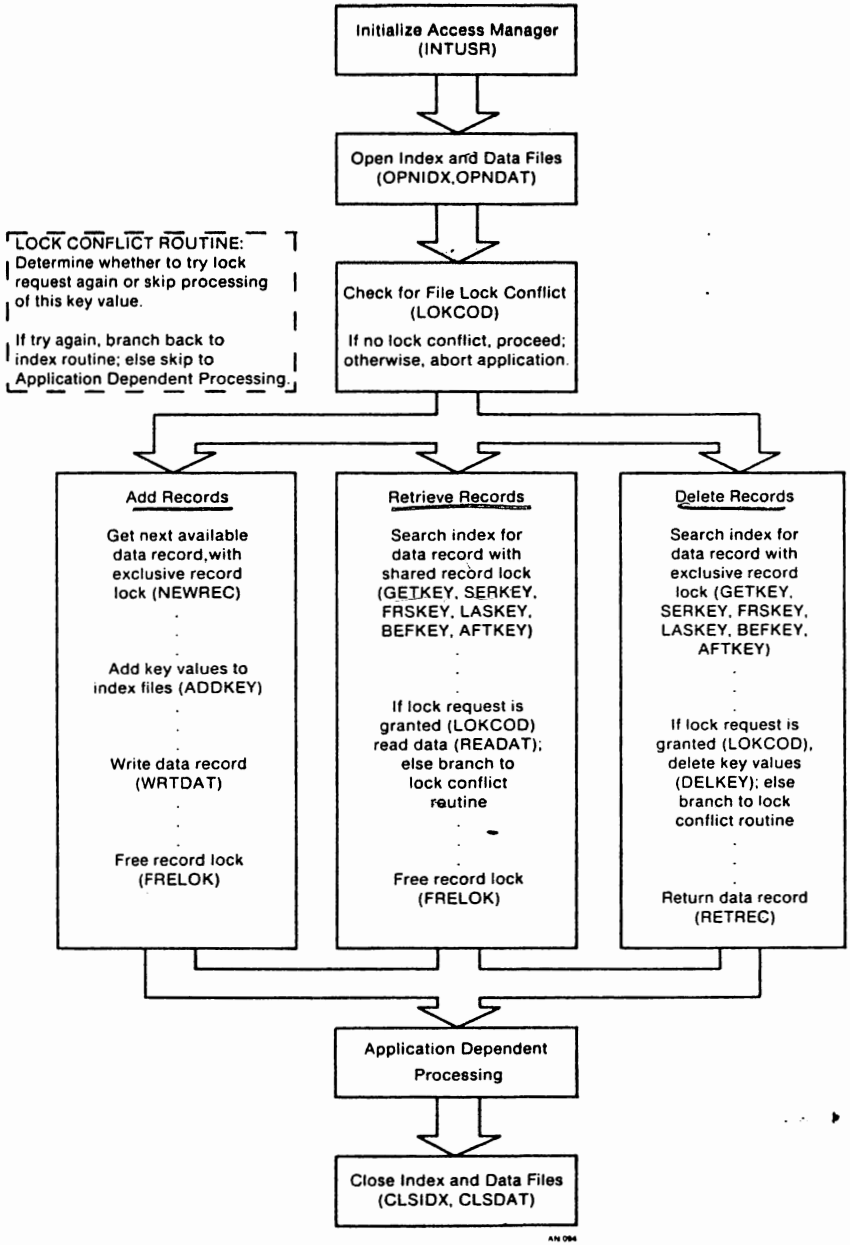

Figure 1-3. Multiuser Program Flow

Note the following points about the multiuser program flow shown in Figure 1-3.

- Your application program can request data record locks as part of the index search or get-next-available-data-record function. This eliminates the need to get a data record number and then request a lock as a separate action.
- If your application program opens a data file with a shared or exclusive data file lock, a call to NEWREC with an exclusive data record lock is always accepted.
- If a lock request is denied as part of an index search, the search function still returns the associated data record number and the key value found in the index (via the IDXVAL$ parameter). Therefore, your application program can use the before and after key functions (BEFKEY and AFTKEY) to skip the locked item.
- The RETREC function, which deletes data records, automatically releases the data record lock if the user holds one. Therefore, do not call the FRELOK function after RETREC.

The importance of data record locking in multiuser environments cannot be overemphasized. If you set shared data record locks when you scan or review a data record and set exclusive record locks when updating, your application program will operate smoothly in multiuser environments.

Remember, Access Manager makes it possible to override the lock settings. Unless your experience and requirements demand you make such overrides, you should strictly adhere to the lock settings. In particular, if a LOKCOD value indicates you did not get the requested lock, be sure your application program can follow an appropriate course of action. Do not ignore the LOKCOD results!

Your Programmer's Guide contains examples illustrating the use of Access Manager to create and maintain a simple data base. Your distribution disk contains the complete source code for these examples.

End of Section 1

Section 2
Function Parameters

Most Access Manager functions are called with parameters that restrict or determine what the function will do. Before reading the function descriptions in Section 3, you should have a basic understanding of these parameters.

2.1 Parameter Types

There are three types of parameters used with Access Manager functions: two-byte integers, character strings, and pointers.

2.1.1 Two-byte Integers

Access Manager treats two-byte integers as unsigned quantities. Most application languages treat them as signed quantities ranging from -32,768 to +32,767. Therefore, if you print a data record number returned from an Access Manager function, you might get a negative result. For example, the largest, unsigned, two-byte quantity is 65,535, which corresponds to -1 as a signed quantity. If you want to display the actual, unsigned value of a two-byte integer, add 65,536 to the quantities in the negative range.

Data record numbers and attributes measuring the number of data records and index file entries are stored in the files as four-byte integer quantities. Two-byte integers span the range from 0 to 65,535. Because most application programs find this range of data record numbers sufficient, Access Manager is designed to use two-byte parameters. Whenever an application program requires the four-byte capacity of Access Manager, the SETDAT and DATVAL functions allow an additional two bytes to be passed to or returned from Access Manager. For example, if your program references multiple data files by a single index file, the two-byte integer returned by an index search function might represent the data record number, while the additional two bytes represent the data file number. SETDAT is called before and DATVAL after other Access Manager functions.

2.1.2 Character Strings

Key values and filenames are always represented as character strings.

2.1.3 Pointers

The READAT and WRTDAT functions use pointers as one of their parameters. Different operating systems sometime require different pointer lengths. Fortunately, your application language automatically sizes pointer parameters correctly.

2.2 Parameter Descriptions

Descriptions of the parameters used with Access Manager functions are summarized in Table 2-1. Specific information on the use and content of each parameter is provided with the individual function descriptions in Section 3. Note that parameter names are suffixed with a dollar ($) or percent (%) sign to indicate the type of value it must contain:

$ = character string value.
% = two-byte integer.

Table 2-1. Access Manager Parameters

Parameter	Description
BUFFER%	This parameter is used only in conjunction with the READAT and WRTDAT functions. It contains a pointer to the input/output buffer area in your application program. When your program reads a data record, Access Manager places it in this buffer area; when it writes a data record, it is written from the buffer.
DLOCK%	[MULTI] A code is passed in this parameter to request a lock on or release a lock from a data file or data record. See the SETLOK function for lock request codes; the FRELOK function for lock release codes.
DRN%	All records in a data file are assigned a unique data record number. Entries in an index file contain the key value of the corresponding record in the data file and its assigned data record number. Thus, the data record number points to the record in the data file with an equal key value. Note that a data record number of zero is considered an error by Access Manager.

Table 2-1. (continued)

Parameter	Description
DUPKEY%	The value passed in this parameter tells Access Manager how duplicate key values in an index file are to be handled. DUPKEY% is used in conjunction with the KEYTYP% parameter. If DUPKEY% is one and KEYTYP% is zero, Access Manager assigns unique sequence numbers to each key value. If KEYTYP% is other than zero, DUPKEY% has no effect.
ERROPT%	The value passed in this parameter tells Access Manager whether or not to trap user errors for handling by your application program. A zero value disables the error-trapping facility; a nonzero value causes Access Manager to trap user errors and make them available to your application program. See the INTUSR function in Section 3 and in Section 4 "Error Codes" for further information.
FILE%	With Access Manager, your application program can process several data files simultaneously. Hence, every data file is assigned a unique file number. The value passed in this parameter tells Access Manager which data file you want to access. Data file numbers used by Access Manager are separate from those used with your application language. Assigning a particular number to an Access Manager data file does not preclude your using the same file number with an application language file.
FILNAME$	Before opening a data file, Access Manager looks in the directory of the disk for the name of the file. The value passed in this parameter must match exactly the name of the data file as it is recorded in the directory.
IDXNAME$	Before opening an index file, Access Manager looks in the directory of the disk for the name of the file. The value passed in this parameter must match exactly the name of the index file as it is recorded in the directory.

Table 2-1. (continued)

Parameter	Description
IDXVAL$	This is the only function parameter that passes a value to your application program. It is used in conjunction with those functions that locate a key value in an index file. At the conclusion of one of these functions, Access Manager places the value of the key it locates in this parameter. Your application program can then use subsequent functions to process that key as required.
KEY%	With Access Manager, your application program can process several index files simultaneously. Hence, every index file is assigned a unique file number. The value passed in this parameter tells Access Manager which index file you want to access. Index file numbers are independent of data file numbers assigned by Access Manager or the application language.
KEYLEN%	To open an index file, Access Manager needs to know the length of the key values in the file. This parameter is used to indicate the number of bytes contained in each key value.
KEYTYP%	To open an index file, Access Manager must know whether the key values are stored in alphanumeric or numeric order. Pass a zero in this parameter to indicate alphanumeric order; a one indicates that key values are stored in numeric order.
KEYVAL$	Entries in an index file contain a key value and associated data record number. To add, delete, or retrieve an entry from an index file, your program must specify the exact value of the key for that entry. This parameter is used to pass the key value to Access Manager.
NBUFS%	[SINGLE] This parameter specifies the number of input/output buffers allocated to your application program. Note that there must be at least three such buffer areas.
NDATF%	[SINGLE] This parameter specifies the maximum number of data files that will ever be open by your program at any given time.

Table 2-1. (continued)

Parameter	Description
NKEYS%	[SINGLE] This parameter specifies the maximum number of index files that will ever be open by your program at any given time.
NNSEC%	[SINGLE] This parameter determines or specifies the length of the records in an index file. Specifically, it refers to the number of 128-byte disk sectors in each index file record.
PROGID%	[MULTI] This parameter contains the unique identifier assigned to each program or task.
RECLEN%	The value of this parameter tells Access Manager the length (that is, number of bytes) of each record in a data file.
TIMOUT%	[MULTI] This parameter specifies the number of seconds within which the background server must respond to the INTUSR function call.

Table 2-2 shows which parameters are used with specific functions. The table also shows whether the parameter value is input (meaning your program passes the value to Access Manager) or output (meaning your program receives the value from Access Manager at the conclusion of the function). Finally, the table shows if the parameter value is required in a multiuser environment, single-user environment, or both.

Table 2-2. Parameter Use Table

Function \ Parameter	BUFFER	DLOCK	DRN	DUPKEY	ERROPT	FILE	FILNAME	IDXNAME	IDXVAL	KEY	KEYLEN	KEYTYP	KEYVAL	NBUFS	NDATF	NKEYS	NNSEC	PROGID	RECLEN	TIMOUT
ADDKEY		Im	Ib			Im				Ib			Ib							
AFTKEY		Im				Ib			Ob	Ib			Ib							
BEFKEY		Im				Ib			Ob	Ib			Ib							
CLSDAT						Ib														
CLSIDX										Ib										
DATVAL																				
DELKEY		Im	Ib			Im				Ib			Ib							
ERADAT		Im				Ib														
ERAIDX										Ib										
ERRCOD																				
FRELOK		Im	Im			Im														
FRSKEY		Im				Ib			Ob	Ib										
GETDFS						Ib														
GETDFU						Ib														
GETKEY		Im				Ib				Ib			Ib							
INTUSR					Ib													Im		Im
LASKEY		Im				Ib			Ob	Ib										
LOKCOD																				
NEWREC		Im				Ib														
NMNODS										Ib										
NOKEYS										Ib										
NXTKEY		Im				Ib			Ob	Ib										
OPNDAT		Im				Ib	Ib												Ib	
OPNIDX				Ib				Ib		Ib	Ib	Ib								
OPRDAT		Im				Ib	Ib												Ib	
OPRIDX				Ib				Ib		Ib	Ib	Ib								
PRVKEY		Im				Ib			Ob	Ib										
READAT	Ib		Ib			Ib														
RETREC		Im	Ib			Ib														
SAVDAT						Ib														
SAVIDX										Ib										
SERKEY		Im				Ib			Ob	Ib			Ib							
SETDAT			Ib																	
SETLOK		Im	Im			Im														
SETUP														Is	Is	Is	Is			
UPDPTR		Im	Ib			Im				Ib			Ib							
WRTDAT	Ib		Ib			Ib														

Im = Input parameter for multiuser environment.
Is = Input parameter for single-user environment.
Ib = Input parameter for single- and multiuser environments.
Ob = Output parameter for single- and multiuser environments.

End of Section 2

Section 3
Function Descriptions

This section contains detailed descriptions of all Access Manager functions. Each function description shows the following information:

- The syntax of the function call as it must be coded in an application program, and the category to which the function is assigned.
- An explanation of the function.
- A list of the parameters required in the command syntax.
- When appropriate, a section of additional comments regarding uses and restrictions placed on the function.
- A descriptive list of the error codes that can be returned by the function. Note that only a brief explanation of the problem is provided. A complete description of each can be found in Section 4, "Error Codes".
- When appropriate and useful, a segment of program code is included as an example of how to use the function in a program.

Access Manager functions return values to your program. The returned value can be an error code, a code indicating the results of the function, or a data record number. Your program must, in all cases, examine the returned value to determine exactly what took place.

Some functions, for example, ADDKEY, DELKEY, and UPDPTR, do not return error codes. The ERRCOD function should be called following any of these functions to test for errors (unless error trapping is not enabled by the ERROPT% parameter). ERRCOD returns a nonzero value if a user error occurs.

Figure 3-1 shows the way Access Manager functions are categorized. The categories are explained in Section 1.

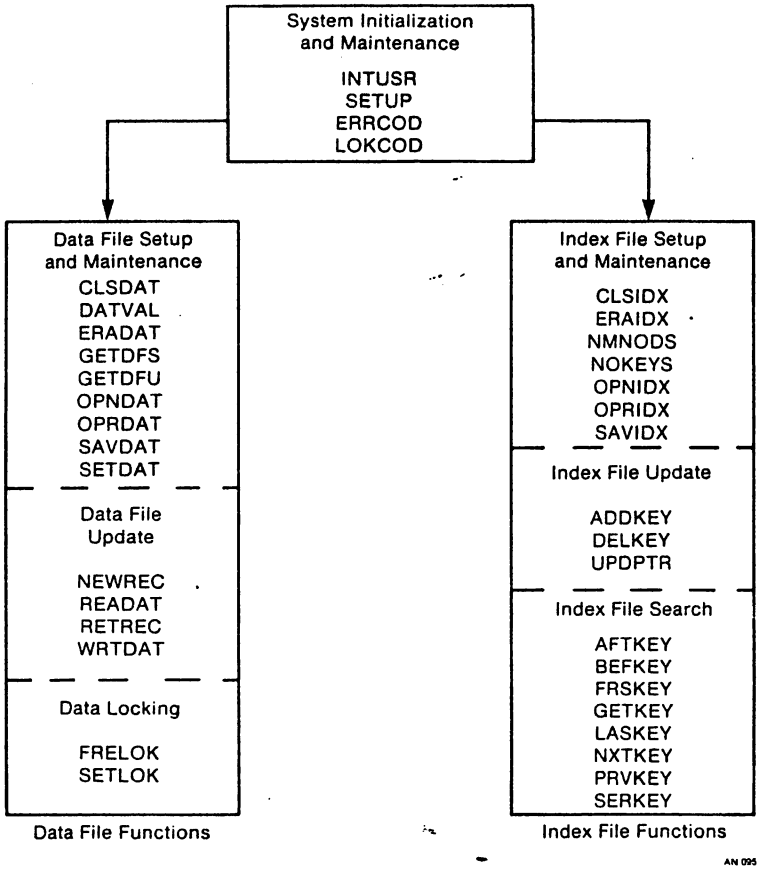

Figure 3-1. Access Manager Functions by Category

ADDKEY Function

Syntax: ADDKEY (KEY%, FILE%, DLOCK%, KEYVAL$, DRN%)

Index File Update Function

Explanation:

ADDKEY adds a new key and associated data record number to an index file and, optionally, assigns sequence numbers to duplicate key values. At the conclusion of the function, a value is returned indicating its success or failure (see Additional Comments below).

Parameters:

KEY% The number of the index file where the key value is added.

FILE% [MULTI] This parameter is ignored in a single-user environment. If you want to ignore it in a multiuser environment, set the DLOCK% parameter to zero. It is the number of the data file associated with the index file specified by the KEY% parameter.

DLOCK% [MULTI] This parameter is ignored in a single-user environment. If you want to ignore it in a multiuser environment, assign a value of zero. It is the type of data record lock requested for the data file specified by the FILE% parameter. See the SETLOK function for a list of acceptable lock codes.

You might encounter situations in a multiuser environment where the associated data record, DRN% parameter, should already have an exclusive record lock before calling ADDKEY (for example, by a previous call to NEWREC). In this case, ADDKEY can be called with a zero DLOCK%, or DLOCK% can be set to two (the code for an exclusive record lock). The lock request should always be granted because, presumably, the same program already holds an exclusive lock. The advantage of the latter approach is that if the requested exclusive record lock is not granted, you might have detected a flaw in your program logic.

KEYVAL$ The key value to be added to the index file. Your application program must ensure that the data record number, DRN% parameter, corresponds to KEYVAL$. If the associated data record number requires more than two bytes, call the SETDAT function immediately before ADDKEY with the higher-order two-byte value as its argument.

If you pass a null string in KEYVAL$, the ADDKEY function takes no action but returns a one unless the associated lock request is denied. A string is null if it has a zero length. Access Manager handles null key values in this manner to simplify coding for multiple key applications. For example, if a data file is indexed by name, number, and zip code, a loop that adds the key values to their respective index files can ignore whether or not a key value is actually present for an index by representing missing values as null strings.

DRN% The data record number associated with KEYVAL$.

Additional Comments:

The ADDKEY function returns the following values to indicate its success or failure:

Table 3-1. ADDKEY Function Values

Value	Meaning
0	KEYVAL$ has been added to the index but the automatic, duplicate-key sequence numbers are exhausted.
1	KEYVAL$ has been successfully added to the index.
2	KEYVAL$ already exists in the index. The index file update has not taken place.
4	The DLOCK% request for the data record (DRN%) in the data file (FILE%) cannot be granted. KEYVAL$ was not added to the index file.

The only assumption Access Manager makes regarding the content of an index file entry is that zero is never used for the data record number. If this happens, user error 36/BD occurs.

In most applications, you should include the key values in both the data file and index file records. This redundancy provides the best protection when or if you have to reconstruct your index files. You should only omit key values from data files if you have an extreme secondary storage space constraint and there are other ways to determine the key values associated with each data record.

Key Value Padding

ADDKEY pads KEYVAL$'s that are less than KEYLEN% bytes with blanks (20H) on the right, and truncates KEYVAL$'s that are too long. However, to ensure proper handling of numeric keys, all numeric KEYVAL$'s must be passed to the Access Manager functions with the exact KEYLEN%.

If you want to store alphanumeric KEYVAL$'s in right-justified form, you must ensure that the KEYVAL$'s are properly justified in view of the truncation of oversized keys.

Duplicate Key Values

There are many situations, such as building an index based on last names, where the key values are not unique. When this occurs, you must append a unique identifier to the key, possibly after truncating the original key value to a prespecified length. This way, the KEYVAL$'s are still stored in the expected order, but there is no conflict between like-valued entries.

If DUPKEY% is one when the index file is opened, Access Manager automatically places a unique, binary sequence number in the last two bytes of the key value. For example, if KEYLEN% is ten and KEYVAL$ equals the string 12345, Access Manager adds three spaces of padding and a two byte sequence number to the end of KEYVAL$ before adding it to the index file. Each set of duplicate key values maintains a separate set of sequence numbers. Thus, there can be up to 65,535 entries in each set of duplicates. Each time another duplicate is added to an existing set of duplicates, the new entry gets the next higher sequence number. Therefore, Access Manager stores duplicate key values in entry order.

You can exhaust the sequence numbers if more than 65,535 identical entries are made. However, if the last entry in a set of duplicates is deleted, this sequence number is reused before the sequence numbers are incremented. When the last available sequence is used for a set of duplicates, ADDKEY returns a value of zero. At this time, the index file must be rebuilt or it will either start to reject duplicates with a return code of two, or enter the duplicates out of sequence. Which of these possibilities takes place depends on the pattern of sequence numbers still in the set of duplicates.

Note: read the DELKEY function description for important information concerning the adverse effect of automatic duplicate keys on the time required to delete a duplicate key value.

Coding Numeric Key Values

If your application requires integer key values, Access Manager provides three alternatives for representing them. In all cases, however, the KEYVAL$ parameter must be a string valued variable. Only the last alternative uses signed, integer key values; the first two alternatives require alphanumeric key values.

The simplest approach to represent an integer key value is to create a string variable which equals the ASCII representation of the integer. With the CBASIC Compiler, the STR$ function automatically performs this conversion. In PL/I, a simple assignment statement of a numeric variable to a CHAR VAR string accomplishes the conversion. In Pascal/MT+, redirected I/O can be used to convert a numeric quantity to a string.

If you use this approach, the resulting strings must be carefully right-justified and padded with blanks or zeros on the left and KEYTYP% should be zero (for an alphanumeric key type). The main disadvantage to this approach is that the key length must be set to the maximum number of digits in the number as opposed to the number of bytes required to store the number in internal binary format.

The two remaining approaches for representing integer key values do take advantage of the compact representation of integers in binary format. One approach treats the key values as signed quantities while the other applies to unsigned integers. In both cases, it is necessary to create a string with characters actually equal to the bytes that comprise the integer quantity. The resulting string is probably an unprintable image because any bit pattern from 00H to FFH might result in each element of the string.

For unsigned integer key values, the CBASIC Compiler function UNSIGNED.INT.KEY$ (see example below) converts an integer quantity into a string equivalent. The function assumes NUMBER is a real, positive quantity; NUMBER can be represented in KEY.LEN% bytes. CHR$ is capable of converting an arbitrary byte value to a string whether or not the byte corresponds to a valid ASCII character.

```
DEF UNSIGNED.INT.KEY$(NUMBER,KEY.LEN%)
STRING TEMP$
INTEGER I%,BYTE%
REAL FACTOR
  TEMP$=""
  FOR I% = 1 TO KEY.LEN%
      FACTOR = INT(NUMBER/256.)
      BYTE% = NUMBER - 256. * FACTOR
      TEMP$ = CHR$(BYTE%) + TEMP$
      NUMBER = FACTOR
  NEXT I%
  UNSIGNED.INT.KEY$ = TEMP$
  RETURN
FEND
```

The preceding function example creates a string whose individual bytes correspond to the bytes necessary to represent NUMBER as an integer with the most significant byte first and the least significant last. As with the first approach, ensure KEYTYP% is zero (for alphanumeric keys).

If you want to treat the integers as signed quantities with negative values preceding positive ones, set KEYTYP% to one and revise the UNSIGNED.INT.KEY$ function so the least significant byte is first and the most significant byte (which includes the sign bit) is last. You must also convert NUMBER to a positive quantity before conversion. With these changes, the conversion function looks like this:

```
DEF SIGNED.INT.KEY$(NUMBER,KEY.LEN%)
STRING TEMP$
INTEGER I%,BYTE%
REAL FACTOR
  IF NUMBER < 0 THEN NUMBER = NUMBER + constant
  TEMP$=""
  FOR I% = 1 TO KEY.LEN%
      FACTOR = INT(NUMBER/256.)
      BYTE% = NUMBER - 256. * FACTOR
      TEMP$ = CHR$(BYTE%) + TEMP$
      NUMBER = FACTOR
  NEXT I%
  SIGNED.INT.KEY$ = TEMP$
  RETURN
FEND
```

In the preceding example, "constant" should be replaced by 256 raised to the KEY.LEN% power. Some example values for constant follow:

Table 3-2. Constant Values

KEY.LEN%	Constant
2	65536.0
3	16777216.0
4	4294975296.0

The preceding values for "constant" also indicate the maximum value plus one for unsigned integer key values. The signed integer key values range from -constant/2 to (constant/2)-1.

For example, let's compare the first and second approaches for integer keys as applied to social security numbers. If SOC.SEC.NO is a real quantity, then an example of using the first approach would be

```
KEYVAL$ = RIGHT$("          " + STR$(SOC.SEC.NO),9)
```

and using the second approach:

```
KEYVAL$ = UNSIGNED.INT.KEY$(SOC.SEC.NO,4)
```

The first approach is faster, but the index file is about 62.5% larger because each entry requires thirteen bytes; nine for the social security number and four for the data record number. The second approach requires only eight bytes.

Note: if signed integer quantities are implemented by setting KEYTYP% to one in OPNIDX, the key value string passed to the index functions must have the least significant byte as the first byte of the string, followed by increasingly significant bytes until the most significant byte is in the last position. Further, the most significant bit of the last byte is treated as the sign; zero if positive, one if negative.

If either the second or third approach is used to compact integer key values, it might be necessary to unpack these strings when they are returned to your program in the output parameter IDXVAL$. The following function unpacks key values prepared according to the second approach:

```
DEF UNPACK.UNSIGNED.INT(IDXVAL$,KEY.LEN%)
REAL TEMP,POWER
INTEGER I%

   POWER = 1
   TEMP = 0
   FOR I% = KEY.LEN% TO 1 STEP -1
       TEMP = TEMP + POWER * ASC(MID$(IDXVAL$,I%,1))
       POWER = POWER * 256.
   NEXT I%
   UNPACK.UNSIGNED.INT = TEMP
   RETURN
FEND
```

The function MID$ returns the string beginning at the position specified by the second parameter with a length given by the third parameter. The function ASC returns the byte value of the first character of its string parameter.

See your Programmer's Guide for information concerning the use of numeric key values with PL/I or Pascal/MT+.

Large Data Files

An Access Manager index file must be contained in one physical disk file. However, this is not true for the data file because the two files are separate entities. Hence, the data file can be spread over more than one disk file. This segmentation of the data file is represented in the index file by the data record numbers associated with the key values.

To use the associated data record numbers to keep track of segmented data files, set the two high-order bytes of the data record number to the segment number and the two low-order bytes to the relative data record number in the segment. For example, the following code inserts the key value corresponding to the ten-thousandth data record of the fifth data file segment:

```
DATA.SEGMENT% = 5
DRN% = 10000
CALL SETDAT(DATA.SEGMENT%)
RET.CODE% = ADDKEY%(KEY%,DATA.SEGMENT%,X.LOCK%, \
  KEYVALUE$,DRN%)
```

An important point concerning segmented data files is setting record locks during index file searches. Because the target segment (that is, data file) is unknown until after the index search, it seems unfeasible to set a record lock at the time the index is searched. However, you can avoid this by always setting the lock on the first segment, regardless of the actual segment where the associated data record is found. This approach is effective because

the lock function uses all four bytes (two for the actual segment and two for the record number) to identify the locked record.

The following examples show how to manage a very large data file and its associated index file. To simplify the discussion, we assume only positive, two-byte logical record numbers (0 through 32,767) are required within each segment.

First, the physical segments comprising the large logical file are opened with shared file locks. For example,

```
FOR SEG.NO% = 1 TO TOTAL.SEG%
  FILE.NO%(SEG.NO%) = OPNDAT(-1,3,FILNAME$(SEG.NO%),RLEN%)
  IF ERRCOD <> 0 THEN CALL ERROR.HANDLER(1)
     IF LOKCOD <> 0 THEN CALL LOCK.CONFLICT(1)
NEXT SEG.NO%
```

To add a new entry, you must first determine the correct physical segment for its placement. The simplest approach is to assume a fixed, upper limit on the active number of records in each segment. The following function returns the segment number to use for the new data file entry:

```
DEF GET.SEGMENT.NO%(ACTIVE.REC.LIMIT%,NO.OF.SEGMENTS%)
    INTEGER SEG.NO%

    FOR SEG.NO% = 1 TO NO.OF.SEGMENTS%
     IF GETDFU(FILE.NO%(SEG.NO%)) < ACTIVE.REC.LIMIT% \
        THEN \
        GET.SEGMENT.NO% = SEG.NO% :\
        RETURN
   NEXT SEG.NO%

   CALL ERROR.HANDLER(2)    REM  No more space
FEND
```

The next code example shows how to set locks and add the new entry to the data and index files. Assume up to 10,000 records can be active in each physical segment:

```
NULL.LOCK% = 0
SHARE.REC.LOCK% = 1
XCLSV.REC.LOCK% = 2

REM  Get next available data record number

SEG.NO% = GET.SEGMENT.NO%(10000,TOTAL.SEG%)
DRN% = NEWREC(FILE.NO%.(SEG.NO%),NULL.LOCK%)

REM  Always set lock on the first segment with a 4-byte DRN
REM  that includes the actual segment number

CALL SETDAT(SEG.NO%)
IF SETLOK%(FILE.NO%(1),XCLSV.REC.LOCK%,DRN%) <> 0 THEN \
   CALL LOCK.CONFLICT(3)

REM  Add key value with upper portion of DRN = SEG.NO

CALL SETDAT(SEG.NO%)
IF ADDKEY(KEY%,FILE.NO%(1),XCLSV.REC.LOCK%,KEYVAL$,DRN%) <> 1 \
   THEN CALL ERROR.HANDLER(4)

REM  Write data record to actual physical segment.  Note
REM  that SETDAT is not used since DRN% is the actual
REM  record number.

IF WRTDAT(FILE.NO%(SEG.NO%),DRN%,BUFFER.PTR%) <> 0 THEN \
   CALL ERROR.HANDLER(5)

REM  Free exclusive record lock

CALL SETDAT(SEG.NO%)
IF FRELOK(FILE.NO%(1),XCLSV.REC.LOCK%,DRN%) <> 0 THEN \
   CALL LOCK.CONFLICT(6)
```

The next code example searches for a data file entry based on the specified KEYVAL$ and applies a shared record lock if it is found.

```
REM  Find, lock, and read data record referenced by KEYVAL%

WAIT.LOOP:
 DRN% = GETKEY(KEY%,FILE.NO%(1),SHARE.REC.LOCK%,KEYVAL$)
 SEG.NO% = DATVAL
 IF LOKCOD <> 0 THEN \
    PRINT "LOCK REQUEST DENIED." :\
    INPUT "ENTER 'W' TO WAIT OR PRESS RET TO SKIP";LINE WAIT$ :\
    IF WAIT$ = "" THEN GOTO SKIP.DATA ELSE GOTO WAIT.LOOP
 IF READAT(FILE.NO%(SEG.NO%),DRN%,BUFFER.PTR%) <> 0 THEN \
    CALL ERROR.HANDLER(7)
```

```
REM  Process data (no updates since shared lock)

    CALL PROCESS.DATA

REM  Free shared lock

    CALL SETDAT(SEG.NO%)
    IF FRELOK(FILE.NO%(1),SHARE.REC.LOCK%,DRN%) <> 0 THEN \
      CALL LOCK.CONFLICT(8)

SKIP.DATA:
```

The final code example demonstrates a deletion from the segmented data file.

```
REM  Find the data file entry by key value

DRN% = GETKEY(KEY%,FILE.NO%(1),XCLSV.REC.LOCK%,KEYVAL$)
SET.NO% = DATVAL

REM  Test if entry is found

IF DRN% = 0 THEN \
  GOTO SKIP.DELETE

REM  Test if exclusive lock obtained

IF LOKCOD <> 0 THEN \
  PRINT "CANNOT DELETE ";KEYVAL$;". CURRENTLY IN USE!" :\
  GOTO SKIP.DELETE

REM  Delete key value

CALL SETDAT(SEG.NO%)
IF DELKEY(KEY%,FILE.NO%(1),XCLSV.REC.LOCK%,KEYVAL$,DRN%) <> 1 \
  THEN CALL ERROR.HANDLER(9)

REM  Free exclusive lock

CALL SETDAT(SEG.NO%)
IF FRELOK(FILE.NO%(1),XCLSV.REC.LOCK%,DRN%) <> 0 THEN \
  CALL LOCK.CONFLICT(10)

REM  Return record to available pool

IF RETREC(FILE.NO%(SEG.NO%),NULL.LOCK%,DRN%) <> 0 THEN \
  CALL ERROR.HANDLER(11)
```

We have assumed that all access to the data file is through the index file. If you must scan the data file without reference to the index file, it is best to perform such a scan with an exclusive file lock, which ensures no unexpected updates to the file during your processing.

Error Codes:

The ADDKEY function can cause these error codes:

Table 3-3. ADDKEY Error Codes

Value	Code	Explanation
21	AE	No directory or disk space available.
30	AN	Index file number (KEY%) is out of range.
36	BD	Key value has a zero data record number.
46	BN	Index file is not open.
147	IC	Lock code (DLOCK%) is out of range.
153	II	Bad parameter value.

Example:

```
DEF EXCEPTION(LOCALE,RETURN.CODE)
    INTEGER LOCALE,RETURN.CODE

    PRINT "Exception at location ";LOCALE
    PRINT "     Error Code:  ";ERRCOD; \
          "      Lock Code:  ";LOKCOD
    PRINT "     Return Code:  ";RETURN.CODE
    STOP
FEND

INPUT "PART NAME:";KEY.VALUE$
DRN% = NEWREC(FILE.NO%,X.LOCK%)
IF ERRCOD <> 0 OR LOKCOD <> 0 THEN \
   CALL EXCEPTION(1,1)
RET.CODE% = ADDKEY(KEY%,INV.FILE%,X.LOCK%,KEY.VALUE$,DRN%)
IF ERRCOD <> 0 OR RET.CODE% <> 1 OR LOKCOD <> 0 THEN \
   CALL EXCEPTION(2,RET.CODE%)
```

EXCEPTION provides a concise method to communicate unexpected results during program development. EXCEPTION is not designed to handle on-line resolution of lock conflicts or error conditions.

AFTKEY Function

Syntax: AFTKEY (KEY%, FILE%, DLOCK%, KEYVAL$, IDXVAL$)

Index File Search Function

Explanation:

AFTKEY returns the data record number associated with the first entry, in key-sequential order, immediately following (that is, strictly greater than) a specific key value. AFTKEY also places the key value it finds in the IDXVAL$ parameter.

Use AFTKEY to move forward through an index file sequentially in key-ascending order if one of the following situations apply:

- your application operates in a multiuser environment.
- you want to update a data file while moving through its index file.

If neither of the preceding situations apply, you will find it efficient to use the NXTKEY function. However, using AFTKEY ensures compatibility between single-user and multiuser environments.

Parameters:

KEY% The number of the index file where the search takes place.

FILE% [MULTI] This parameter is ignored in a single-user environment. It is the number of the data file referenced by the KEY% parameter.

DLOCK% [MULTI] This parameter is ignored in single-user environments. It specifies the type of data record lock requested for the data file referenced by the FILE% parameter. If the requested lock for the data record that AFTKEY found cannot be granted, LOKCOD returns a nonzero value. See the SETLOK function for a list of acceptable lock codes.

KEYVAL$ The key value that precedes the one being sought. Note that KEYVAL$ does not have to exist in the index for AFTKEY to function. KEYVAL$ serves only as a reference value for seeking the actual index entry.

IDXVAL$ This is an OUTPUT parameter. Access Manager places the key value found in the index file in IDXVAL$ at the conclusion of the function. If no index entry is found with a key value greater than KEYVAL$, AFTKEY returns zero, and IDXVAL$ is set to all blanks (which is not the same as a null string).

Before calling the AFTKEY function, IDXVAL$ must be at least as long as the key length specified for the KEY% file. Normally, it is only necessary to initialize IDXVAL$ once at the beginning of a program. Notice that KEYVAL$ and IDXVAL$ must not be the same variable.

Additional Comments:

To determine the two high-order bytes of the associated data record number, call the DATVAL function immediately following AFTKEY.

Error Codes:

The AFTKEY function can cause the error codes listed in Table 3-4.

Table 3-4. AFTKEY Error Codes

Value	Code	Explanation
30	AN	Index file number (KEY%) is out of range.
35	BC	IDXVAL$ too short to contain key value.
46	BN	Index file is not open.
147	IC	Lock code (DLOCK%) is out of range.
153	II	Bad parameter value.

Example:

The following example listing shows many of the index and data file functions. It creates a physically ordered data file from an existing indexed data file. Also, the index file is updated in place to reflect the new data record positions.

```
REM
REM   External AM80 Declarations
REM

%INCLUDE AM80EXTR.BAS

REM
REM Error Handling Routine
REM

DEF ERROR.HANDLER(LOCALE)
        INTEGER LOCALE

    PRINT
    PRINT "Fatal ERROR at Locale ";LOCALE
    PRINT "      ERROR Code ";ERRCOD
    PRINT "      Return Code ";RET.CODE%
    STOP
FEND

DEF TEMP.LOCK
    PRINT "Temporary Sort File ";FIL.NAME$; \
          " cannot be locked!" : \
    DUMMY% = CLSDAT(OLD.FILE%%) : \
    STOP
FEND

REM
REM System Parameters
REM

YES% = -1
NO% = 0

NBUF% = 6
NKEYS% = 1
NNSEC% = 4
NDATF% = 2

X.FILE% = 4
N.LOCK% = 0

PROGID% = -1
ERROR.TRAP% = YES%
TIME.OUT.TEST% = 3

REM
REM System Initialization
REM
```

Listing 3-1. AFTKEY Function Program Code

```
PROGID% = INTUSR(PROGID$,ERROR.TRAP%,TIME.OUT.TEST%)
IF ERRCOD <> 0 THEN \
    CALL ERROR.HANDLER(1)
IF SETUP(NBUF%,NKEYS%,NNSEC%,NDATF% <> 0 THEN \
    CALL ERROR.HANDLER(2)

REM
REM Data File Set Up
REM

INPUT "Enter data filename & record length:"; \
    FIL.NAME$,REC.LEN%
OLD.FILE% = -1
NEW.FILE% = -1
OLD.FILE% = OPNDAT(OLD.FILE%,X.FILE%,FILE.NAME$,REC.LEN%)
IF ERRCOD <> 0 THEN \
    CALL ERROR.HANDLER(3)
IF LOKCOD <> 0 THEN \
    PRINT "Data File ";FIL.NAME$;" cannot be locked at"; \
          " this time.  Try later!" :\
    STOP

FIL.NAME$ = "SORT.$$" + RIGHT$(STR$(PROGID%),1)
NEW.FILE% = OPNDAT(NEW.FILE%,X.FILE%,FIL.NAME$,REC.LEN%)

REM
REM Check for left over SORT.$$?  If lock granted, erase
REM then reopen file to ensure fresh data file.  See
REM Section 4 for description of User Error Codes.
REM

IF ERRCOD <> 0 AND ERRCOD <> 53 THEN \
    CALL ERROR.HANDLER(4)

IF LOKCOD <> 0 THEN \
    CALL TEMP.LOCK

IF ERADAT(NEW.FILE%,N.LOCK%) <> 0 THEN \
    CALL ERROR.HANDLER(14)

NEW.FILE% = -1
NEW.FILE% = OPNDAT(NEW.FILE%,X.FILE%,FIL.NAME$,REC.LEN%)
IF ERRCOD <> 0 THEN \
    CALL ERROR.HANDLER(15)
IF LOKCOD <> 0 THEN \
    CALL TEMP.LOCK

REM
REM I/O Buffer Set Up
REM
```

Listing 3-1. (continued)

```
REM                  1         2         3         4       4
FILLER$ = "12345678901234567890123456789012345678901234567890123456789012345678"
IO.BUFFER$ = ""
NO.FILLERS% = INT%((REC.LEN% + 47) / 48)
FOR I% = 1 TO NO.FILLERS%
    IO.BUFFER$ = IO.BUFFER$ + FILLER$
NEXT I%
BUFFER.PTR% = SADD(IO.BUFFER$) + 2
REM  "+2" because each string has a 2-byte length header

REM
REM  Index File Set Up
REM

INPUT "Enter index name, key length, & type:"; IDX.NAME$, \
    KEY.LEN%,KEY.TYPE%
KEY% = -1
KEY% = OPNIDX(KEY%,IDX.NAME$,KEY.LEN%,KEY.TYPE%,0)
IF ERRCOD <> 0 THEN \
    CALL ERROR.HANDLER(4)
IDX.ENTRY$ = LEFT$(FILLER$,KEY.LEN%)

REM
REM  Get the First Key Value
REM

DRN% = FRSKEY(KEY%,OLD.FILE%,N.LOCK%,IDX.ENTRY$)
DRN2% = DATVAL
IF ERRCOD <> 0 THEN \
    CALL ERROR.HANDLER(5)

REM
REM Loop over key values until index is exhausted, writing
REM   data to new file in key value order
REM

WHILE DRN% <> 0 OR DRN2% <> 0

REM  Read old data file

    CALL SETDAT(DRN2%)
    IF READAT(OLD.FILE%,DRN%,BUFFER.PTR%) <> 0 THEN \
        CALL ERROR.HANDLER(6)

REM  Get next record in new file

    SORT.DRN% = NEWREC(NEW.FILE%,N.LOCK%)
    SORT.DRN2% = DATVAL
    IF ERRCOD <> 0 THEN \
        CALL ERROR.HANDLER(7)
```

Listing 3-1. (continued)

```
REM  Write out new record (in key value order)

    CALL SETDAT(SORT.DRN2%)
    IF WRTDAT(NEW.FILE%,SORT.DRN%,BUFFER.PTR%) <> 0 THEN \
        CALL ERROR.HANDLER(8)

REM  Update pointer in index file to reflect sorted order

    CALL SETDAT(SORT.DRN2%)
    RET.CODE% = UPDPTR(KEY%,NEW.FILE%,N.LOCK%,IDX.ENTRY$, \
        SORT.DRN%)
    IF ERRCOD <> 0 THEN \
        CALL ERROR.HANDLER(9)

REM  Get next key value

    TARGET$ = LEFT$(IDX.ENTRY$,KEY.LEN%)
    DRN% = AFTKEY(KEY%,OLD.FILE%,N.LOCK%,TARGET$,IDX.ENTRY$)
    DRN2% = DATVAL
    IF ERRCOD <> 0 THEN \
        CALL ERROR.HANDLER(10)
WEND

REM
REM  Close files
REM

IF CLSDAT(NEW.FILE%) <> 0 THEN \
    CALL ERROR.HANDLER(11)
IF CLSDAT(OLD.FILE$) <> 0 THEN \
    CALL ERROR.HANDLER(12)
IF CLSIDX(KEY%) <> 0 THEN \
    CALL ERROR.HANDLER(13)

REM
REM  Sign-Off Message
REM

PRINT "Last record written:  ";SORT.DRN2%;SORT.DRN%
STOP
```

Listing 3-1. (continued)

BEFKEY Function

Syntax: BEFKEY (KEY%, FILE%, DLOCK%, KEYVAL$, IDXVAL$)

Index File Search Function

Explanation:

BEFKEY returns the data record number assigned to the index entry immediately preceding (that is, strictly less than) a key value you provide. BEFKEY also places the value of the key it finds in the IDXVAL$ parameter.

Use BEFKEY to move backward through an index file sequentially in key-descending order if either of the following situations apply:

- your application operates in a multiuser environment.
- you want to update the index file while moving through it.

If neither of the preceding situations apply, you will find it more efficient to use the PRVKEY function. Use BEFKEY to maintain compatibility between single-user and multiuser environments.

Parameters:

KEY% The number of the index file where the search takes place.

FILE% [MULTI] This parameter is ignored in a single-user environment. It is the number of the data file referenced by the KEY% parameter.

DLOCK% [MULTI] This parameter is ignored in single-user environments. It specifies the type of data record lock requested for the data file referenced by the FILE% parameter. If the requested lock for the data record BEFKEY found cannot be granted, LOKCOD returns a nonzero value. See the SETLOK function for a list of acceptable lock codes.

KEYVAL$ The key value immediately following the one being sought. Note that KEYVAL$ does not have to exist in the index for BEFKEY to function. KEYVAL$ serves only as a reference value for seeking the actual index entry.

IDXVAL$ This is an OUTPUT parameter. Access Manager places the key value found in the index file in IDXVAL$ at the conclusion of the function. If no index entry is found with a key value less than KEYVAL$, BEFKEY returns zero, and IDXVAL$ is set to all blanks (which is not the same as a null string).

Before calling the BEFKEY function, IDXVAL$ must be at least as long as the key length specified for the KEY% file. Normally, it is only necessary to initialize IDXVAL$ once at the beginning of a program. Also notice that KEYVAL$ and IDXVAL$ must not be the same variable.

Additional Comments:

To determine the two high-order bytes of the associated data record number, call the DATVAL function immediately after BEFKEY.

Error Codes:

Table 3-5 lists the BEFKEY function error codes.

Table 3-5. BEFKEY Error Codes

Value	Code	Explanation
30	AN	Index file number (KEY%) is out of range.
35	BC	IDXVAL$ too short to contain key value.
46	BN	Index file is not open.
147	IC	Lock code (DLOCK%) is out of range.
153	II	Bad parameter value.

Example:

In this example, BEFKEY is used to determine the sequence number automatically appended to the key value by the ADDKEY function. This assumes the DUPKEY% parameter is set to one in the call to OPNIDX. Further assumptions are that this is a single-user environment and the function SPACE$ has already been defined and returns a string of blanks.

```
INPUT "Enter key value:";KEYVAL$
IF ADDKEY(KEY%,0,0,KEYVAL$,DRN%) <> 1 THEN \
   CALL ERROR.HANDLER(1)

IF LOKCOD <> 0 THEN CALL ERROR.HANDLER(2)

TARGET$ = LEFT$(KEYVAL$+SPACE$(KEYLEN%),KEYLEN%-2) + \
   CHR$(0FFH)+CHR$(0FFH)

IDXVAL$ = SPACE$(KEYLEN%)

IF BEFKEY(KEY%,0,0,TARGET$,IDXVAL$) <> DRN% THEN \
   CALL ERROR.HANDLER(3)

IF ERRCOD <> 0 THEN CALL ERROR.HANDLER(4)

SEQUENCE.NO$ = RIGHT$(IDXVAL$,2)
```

The approach used in this example works because the last key added to the index has the largest sequence number less than 0FFFFH.

CLSDAT Function

Syntax: CLSDAT (FILE%)

Data File Setup and Maintenance Function

Explanation:

CLSDAT closes a data file. If the data file is closed successfully, CLSDAT returns a zero at the conclusion of the function; otherwise, a nonzero user error code is returned.

Parameters:

FILE% The number of the closed data file.

Additional Comments:

If you make any changes to a data file, your application program must call the CLSDAT or SAVDAT function to force the updates to the disk file. If you do not do this, the integrity of the data file can be disrupted because the Access Manager header record could be incorrect. Further, the operating system might be holding data in internal buffers. CLSDAT and SAVDAT force the updated header record to the disk and flush the operating system buffers.

[MULTI] If other programs are using the data file, CLSDAT actually performs a save operation. The data file is still available to the other users. Because Access Manager opens files in a locked mode, a file is not accessible to users outside of Access Manager unless all programs that opened the file subsequently closed it.

If CLSDAT is called prior to program termination, you should also call the FRELOK function to release whatever type of data file lock you requested at the time the file was opened.

Error Codes:

If a data file is not properly closed or saved after it is updated, Access Manager issues user error 70/DF the next time that data file is opened. In such cases, the data file must be reconstructed.

Table 3-6 lists the CLSDAT function error codes.

Table 3-6. CLSDAT Error Codes

Value	Code	Explanation
55	CG	Data filename not in directory.
60	CL	Data file number (FILE%) is out of range.
74	DJ	Data file (FILE%) is inactive.
177	KA	Lock code (DLOCK%) is out of range
183	KG	Bad parameter value.

Example:

```
IF CLSDAT(FILE.NO%) <> 0 THEN \
   PRINT "Error while closing data file."
```

CLSIDX Function

Syntax: CLSIDX (KEY%)

Index File Setup and Maintenance Function

Explanation:

CLSIDX closes an index file. If the index file is closed successfully, CLSIDX returns a zero at the conclusion of the function; otherwise, a nonzero user error code is returned.

Parameters:

KEY% The number of the closed index file.

Additional Comments:

If you make any changes to an index file, your application program must call the CLSIDX or SAVIDX function to force the updates to the disk file. If you do not do this, the integrity of the index file can be disrupted because some updated nodes might still be in an I/O buffer and/or the header record could be incorrect.

[MULTI] If other programs are still using the index file, CLSIDX actually performs a save operation. The index file is still available to the other users. Because Access Manager opens files in a locked mode, a file is not accessible to users outside of Access Manager unless all programs that opened the file subsequently closed it.

Error Codes:

If an index file is not properly closed or saved after it is updated, Access Manager issues user error 40/BH the next time that index file is opened. In such cases, the index file must be reconstructed.

Table 3-7 lists the CLSIDX function error codes.

Table 3-7. CLSIDX Error Codes

Value	Code	Explanation
21	AE	Disk or directory full.
25	AI	Index filename not found in directory.
30	AN	Index file number (KEY%) is out of range.
44	BL	Index file (KEY%) is inactive.
46	BN	Index file (KEY%) is not open.
153	II	Bad parameter value.

Example:

```
IF CLSIDX(KEY.NO%) <> 0 THEN \
   CALL ERROR.HANDLER
```

DATVAL Function

Syntax: DATVAL

Data File Setup and Maintenance Function

Explanation:

DATVAL returns the high-order two bytes of the data record number. DATVAL can be called following any Access Manager function that returns a data record number as a function value (such as NEWREC) or a size attribute of the data file (GETDFS, GETDFU, or NOKEYS).

Parameters:

The DATVAL function is called without parameters.

Additional Comments:

If your application program does not require more than 65,535 data records or index file entries, there is no need to call DATVAL. However, there is no harm in doing so under these circumstances because a value of zero is returned.

Error Codes:

The DATVAL function does not cause user errors.

Example:

Note that DATVAL is called immediately after the NEWREC function in the following example. DRN2% contains the most significant two bytes of the data record number, and DRN% contains the least significant two bytes.

```
DLOCK% = 2      REM Exclusive record lock
DRN% = NEWREC(FILE.NO%,DLOCK%)
DRN2% = DATVAL
IF ERRCOD <> 0 THEN \
     CALL ERROR.HANDLER(3)
IF LOKCOD <> 0 THEN \
     CALL LOCK.CONFLICT(3)
```

DELKEY Function

Syntax: DELKEY (KEY%, FILE%, DLOCK%, KEYVAL$, DRN%)

Index File Update Function

Explanation:

DELKEY removes a key and its data record number from a specified index file. At the conclusion of the function, DELKEY returns a value indicating its success or failure (see Additional Comments below).

Parameters:

KEY% The number of the index file where the key value and data record number are removed.

FILE% [MULTI] This parameter is ignored in single-user environments. It is the number of the data file referenced by the KEY% parameter.

DLOCK% [MULTI] This parameter is ignored in single-user environments. It contains the code to specify the type of data record lock requested for the data file referenced by the FILE% parameter. See the SETLOK function for a list of acceptable lock codes.

To ignore locking protocols in a multiuser environment, assign DLOCK% a value of zero. This procedure is not normally recommended.

KEYVAL$ The value of the key record to be deleted from the index file.

If you pass a null string in the KEYVAL$ parameter, the DELKEY function takes no action but returns a value of one unless the requested DLOCK% is denied. This simplifies coding in multiple key applications. A loop designed to delete all key values for a given data record from their respective index files does not have to test for nonexistent keys as long as they appear as null strings.

DRN% The data record number associated with the key value contained in KEYVAL$. Specifically, this is the number of the record to be deleted from the index file. Access Manager checks the data record number associated with KEYVAL$ before making the deletion.

Additional Comments:

DELKEY returns one of the following values at its conclusion to indicate its success or failure:

Table 3-8. DELKEY Function Values

Value	Meaning
0	KEYVAL$ was not found in the index.
1	KEYVAL$ was successfully deleted from the index file.
2	The data record number in the index file containing KEYVAL$ does not agree with the DRN% parameter. The index entry was not deleted.
4	The DLOCK% request on the data record given by DRN% for the data file specified by FILE% could not be granted. The index entry was not deleted.

If an index file is opened with a DUPKEY% parameter of one (implying automatic suffixing of key values), DELKEY ignores the last two bytes of KEYVAL$ and searches for an entry matching the first KEYLEN%-2 bytes, and for which the data record number equals DRN%. If a match is found, the entry is deleted from the index file. If the key value in KEYVAL$ is less than KEYLEN% bytes in length, it is padded with blanks before the last two bytes are truncated.

Because Access Manager must search a set of duplicates for the one with a matching data record number, the number of disk accesses required to delete a duplicate key value can be excessive for a very large set of identical keys.

If you foresee large sets of duplicate key values (several hundred or thousands of identical values) and a regular need to delete members of these sets, you can avoid longer delete times by not using the Access Manager automatic duplicate feature. Instead, you should append your own unique suffix to these keys. For example, you might append a data record number to the end of the key value or an associated unique key value. If you append your own suffix, set the DUPKEY% parameter in the OPNIDX function to zero. When you delete one of these key values, you can construct the exact composite key (identical portion plus unique suffix) to use in a regular deletion operation. No special Access Manager search of the duplicate set is required.

Error Codes:

The DELKEY function error codes are listed in Table 3-9.

Table 3-9. DELKEY Error Codes

Value	Code	Explanation
30	AN	Index file number (KEY%) is out of range.
46	BN	Index file is not open.
147	IC	Lock code (DLOCK%) is out of range.
153	II	Bad parameter value.

Example:

In this example, SETDAT passes the two high-order bytes of the associated data record number.

```
CALL SETDAT(DRN2%)
IF DELKEY(KEY%,FILE.NO%,LOCK%,KEY.VALUE$,DRN%) <> 1 \
   THEN PRINT "Unsuccessful Deletion"
```

ERADAT Function

Syntax: ERADAT (FILE%, DLOCK%)

Data File Setup and Maintenance Function

Explanation:

ERADAT erases a data file. Note that the data file must have been previously opened using either the OPNDAT or OPRDAT function. At the conclusion of the ERADAT function, a zero value is returned if the erasure was successful; otherwise, a nonzero user error code results.

Parameters:

FILE% The number of the erased data file. If the data file is not open when ERADAT is called, an error results.

DLOCK% [MULTI] Specifies the type of data file lock operation requested for the data file to be erased. If the value is set to zero when ERADAT is called, all data file lock operations are ignored. Use extreme care when calling ERADAT without an exclusive data file lock. See the SETLOK function for a list of acceptable lock codes.

If a file lock is requested and granted, the data file is erased. If the file lock request cannot be granted, the data file is not erased. Your application program must examine the contents of LOKCOD immediately after calling ERADAT to determine if the lock request was granted.

Additional Comments:

Use the ERADAT function with care to prevent accidental loss of a data file.

If your application program creates temporary data files, ERADAT can be used to delete them when necessary.

If your application program requires that a data file be new, successive calls to OPRDAT, ERADAT, and then OPNDAT ensure this.

Error Codes:

Access Manager returns a value in ERADAT to indicate the results of the operation. If no user errors occur, a zero is returned. The ERADAT function can cause any of the following user errors:

Table 3-10. ERADAT Error Codes

Value	Code	Explanation
60	CL	Data file number (FILE%) is out of range.
175	JO	The data file (FILE%) is inactive.
176	K@	File name not found in directory.
177	KA	Lock code (DLOCK%) is out of range.
183	KG	Bad parameter value.

Example:

In the following example, the data file is assumed to be open. The arguments of the ERROR.HANDLER and LOCK.CONFLICT functions indicate from where the functions were called.

```
DLOCK% = 4      REM  Exclusive data file lock
IF ERADAT(FILE.NO%,DLOCK%) <> 0 THEN \
     CALL ERROR.HANDLER(2)
IF LOKCOD <> 0 THEN \
     CALL LOCK.CONFLICT(2)
```

ERAIDX Function

Syntax: ERAIDX (KEY%)

Index File Setup and Maintenance Function

Explanation:

ERAIDX erases an index file. The index file must have been opened previously using the OPNIDX or OPRIDX function. ERAIDX returns a zero at the conclusion of the function if the erasure was successful; otherwise, a nonzero user error code results.

Parameters:

KEY%	The number of the erased index file. If the index file is not open when ERAIDX is called, an error results.

Additional Comments:

Use the ERAIDX function with care to prevent accidental loss of an index file.

If your application program creates temporary index files, ERAIDX can be used to delete them when necessary.

If your application program requires that an index file be new, successive calls to OPRIDX, ERAIDX, and OPNIDX will initialize the file correctly.

Error Codes:

ERAIDX returns the error code as its function value. If ERAIDX returns zero, erasure of the index file is successful; otherwise, a user error has occurred. The ERAIDX function can cause the user errors listed in Table 3-11.

Table 3-11. ERAIDX Error Codes

Value	Code	Explanation
30	AN	Index file number (KEY%) is out of range.
46	BN	Index file is not open.
145	IB	The index file (KEY%) is inactive.
146	IC	Index file name not found in directory.
153	II	Bad parameter value.

ERRCOD Function

Syntax: ERRCOD

System Initialization and Maintenance Function

Explanation:

ERRCOD returns the current value of the user error code.

ERRCOD returns a zero if the function completes execution without a user error occurring. On the other hand, if a user error DOES occur and the ERROPT% parameter (see INTUSR function) was passed with a nonzero value, ERRCOD returns the appropriate error code value.

Parameters:

The ERRCOD function is called without parameters.

Additional Comments:

The value of ERRCOD does not change until a subsequent index or data file function is called. Consequently, if a function causes a user error, later calls to ERRCOD without intervening calls to other functions return the same error code value. ERRCOD clears to zero only when an Access Manager function executes without a user error.

Access Manager checks for two types of errors: user errors and failures of internal consistency. (Internal consistency errors are explained in Section 4.) When internal consistency errors occur, Access Manager cannot trap them, so it routes an error message to the console and returns control to the operating system.

Error Codes:

Calls to ERRCOD do not cause user errors.

Example:

```
RET.CODE% = ADDKEY(KEY.NO%,FILE.NO%,LOCK.REQ%,KEY.VALUE$,DRN%)
IF ERRCOD <> 0 THEN CALL ERROR.HANDLER
IF RET.CODE% <> 1 THEN CALL ADD.KEY.PROBLEM
```

FRELOK Function

Syntax: FRELOK (FILE%, DLOCK%, DRN%)

Data Locking Function

Explanation:

FRELOK releases (or, frees) an existing lock on a data file or data record. If the lock is released successfully, the FRELOK function returns a zero at its conclusion; otherwise, a one is returned. This function has no effect if called in a single-user environment.

Parameters:

FILE%	The number of the data file from which you want to release the lock.
DLOCK%	Table 3-12 lists the codes you can use to request release of a data file lock. The appropriate code is passed via this parameter.
DRN%	The number of the data record from which the lock is to be released.

Additional Comments:

FRELOK attempts to remove a specified type of lock held by the calling program (see PROGID% parameter under the INTUSR function). To release a lock, set the DLOCK% parameter to the appropriate value shown in Table 3-12. Note that if the specified file or record does not have a lock set when FRELOK is called, no action takes place and LOKCOD is set to one.

Table 3-12. Data File/Record Lock Release Requests

DLOCK% Value	Unlock Request
0	No action, but LOKCOD is set to zero.
1	Release shared lock on DRN%.
2	Release exclusive lock on DRN%.
3	Release shared file lock on FILE%.
4	Release exclusive file lock on FILE%.
5	Release either shared or exclusive record lock on DRN%.
6	Release all locks held by the calling program (PROGID%) on the specified FILE%.
7	Release all locks held by the calling program (PROGID%) on all files with numbers greater than or equal to FILE%.

It is far more efficient to release each record lock after the record is processed than to release all record locks at the same time. While Access Manager has no practical limit on the number of record locks held simultaneously, a large number of locks forces some of the lock information out to disk. This results in slower processing.

Error Codes:

Table 3-13 lists the error codes for FRELOK function.

Table 3-13. FRELOK Error Codes

Value	Code	Explanation
52	CD	Data record number (DRN%) is zero or beyond end of file.
60	CL	Data file number (FILE%) is out of range.
177	KA	Lock code (DLOCK%) is out of range.
183	KG	Bad parameter value.

Example:

```
IF FRELOK(FILE.NO%,5,DRN%) <> 0 THEN \
   PRINT "No record lock was held on ";DRN%: \
   PRINT "by this user."
```

FRSKEY Function

Syntax: FRSKEY (KEY%, FILE%, DLOCK%, IDXVAL$)

Index File Search Function

Explanation:

The FRSKEY function locates the first entry in an index file. This function returns the data record number assigned to the first entry and also returns its key value in the IDXVAL$ parameter.

Parameters:

KEY% The number of the index file where you want to search for the first index entry.

FILE% [MULTI] This parameter is ignored in a single-user environment. It is the number of the data file referenced by the index file specified by the KEY% parameter.

DLOCK% [MULTI] This parameter is ignored in single-user environments. It specifies the type of data record lock requested for the data file referenced by the FILE% parameter. If the requested lock for the data record that FRSKEY found cannot be granted, LOKCOD returns a nonzero value. See the SETLOK function for a description of lock codes.

IDXVAL$ This is an OUTPUT parameter. Access Manager places the key value found in the index file in IDXVAL$. If the index is empty, FRSKEY returns a zero and IDXVAL$ is set to all blanks (which is not the same as a null string).

Before calling the FRSKEY function, IDXVAL$ must be at least as long as the key length specified for the KEY% file or user error 35/BC results.

Additional Comments:

To determine the two high-order bytes of the associated data record number, call the DATVAL function immediately after FRSKEY.

Error Codes:

The FRSKEY function can cause the error codes listed in Table 3-14.

Table 3-14. FRSKEY Error Codes

Value	Code	Explanation
30	AN	Index file number (KEY%) is out of range.
35	BC	IDXVAL$ too short to contain key value.
46	BN	Index file is not open.
147	IC	Lock code (DLOCK%) is out of range.
153	II	Bad parameter value.

Example:

```
REC.LOK% = 2      REM  Exclusive Record Lock Request
DRN% = FRSKEY(KEY.NO%,FILE%,REC.LOK%,IDXVAL$)
SEG.NO% = DATVAL
IF ERRCOD <> 0 THEN CALL ERROR.HANDLER
IF LOKCOD <> 0 THEN CALL LOCK.CONFLICT
IF DRN% = 0 AND SEG.NO% = 0 THEN \
   PRINT "Index file is empty."
```

GETDFS Function

Syntax: GETDFS (FILE%)

Data File Setup and Maintenance Function

Explanation:

GETDFS returns a count of the records in a data file. The count includes the number of records in current use and those available for use.

Parameters:

FILE% The number of the data file where the record count takes place.

Additional Comments:

The record count returned by the GETDFS function includes the header record and any other records automatically reserved by the NEWREC function. This ensures that the first available data record begins on or after the 129th byte of the data file, as required by Access Manager.

To determine the two high-order bytes of the logical data file size, call the DATVAL function immediately after GETDFS. Also see the GETDFU function.

Error Codes:

Table 3-15 lists the GETDFS Error Codes.

Table 3-15. GETDFS Error Codes

Value	Code	Explanation
60	CL	Data file number (FILE%) is out of range.
183	KG	Bad parameter value.

Example:

If the file is certain to contain less than 65,535 records, the actual number of kilobytes used by the data file can be computed as follows:

```
IF GETDFS(FILE.NO%) < 0 THEN \
   TOTREC = GETDFS(FILE.NO%) + 65536.  \
ELSE \
   TOTREC = GETDFS(FILE.NO%)
KILOBYTES% = INT%((TOTREC * REC.LEN% + 1023.) / 1024.)
```

where REC.LEN% is the record length of the file specified by FILE.NO%. Note how the signed integer quantity is converted to a positive real quantity.

If the file might contain more than 65,535 records, the number of kilobytes is given by

```
IF GETDFS(FILE.NO%) < 0 THEN \
   TOTREC = GETDFS(FILE.NO%) + 65536.  \
ELSE \
   TOTREC = GETDFS(FILE.NO%)
TOTREC = TOTREC + (65536. * DATVAL)
KILOBYTES% = INT%((TOTREC * REC.LEN% + 1023.) / 1024.)
```

In this case, DATVAL returns the two high-order bytes of the data file size. If the size is less than 65,536, DATVAL returns zero.

GETDFU Function

Syntax: GETDFU (FILE%)

Data File Setup and Maintenance Function

Explanation:

GETDFU returns a count of records in use in a data file.

The difference between the counts returned by GETDFS and GETDFU represents the data records returned for reuse, plus the header record, plus (for record lengths less than 128 bytes) records that share the first 128 bytes with the header record.

Parameters:

FILE% The number of the data file where the record count takes place.

Additional Comments:

To determine the two high-order bytes of the in-use record count, call the DATVAL function immediately after GETDFU.

Error Codes:

Table 3-16 lists the GETDFU error codes.

Table 3-16. GETDFU Error Codes

Value	Code	Explanation
60	CL	Data file number (FILE%) is out of range.
183	KG	Bad parameter value.

Example:

GETDFU and GETDFS can be used together to compute the fraction of data records in active use, as shown in this example:

```
IF GETDFU(FILE.NO%) < 0 THEN \
   ACTIVE.RECORDS = GETDFU(FILE.NO%) + 65536.  \
ELSE \
   ACTIVE.RECORDS = GETDFU(FILE.NO%)
ACTIVE.RECORDS = ACTIVE.RECORDS + (65536. * DATVAL)

IF GETDFS(FILE.NO%) < 0 THEN \
   TOTAL.RECORDS = GETDFS(FILE.NO%) + 65536.  \
ELSE \
   TOTAL.RECORDS = GETDFS(FILE.NO%)
TOTAL.RECORDS = TOTAL.RECORDS + (65536. * DATVAL)

UTILIZATION = ACTIVE.RECORDS / TOTAL.RECORDS
```

GETKEY Function

Syntax: GETKEY (KEY%, FILE%, DLOCK%, KEYVAL$)

Index File Search Function

Explanation:

GETKEY returns the data record number associated with a specific key value in an index file. GETKEY locates the entry in the index file that exactly matches a key value you provide in the KEYVAL$ parameter.

Parameters:

KEY%	The number of the index file where the key value/data record number can be located.
FILE%	[MULTI] This parameter is ignored in a single-user environment. It is the number of the data file referenced by the KEY% parameter.
DLOCK%	[MULTI] This parameter is ignored in single-user environments. It specifies the type of data record lock requested for the data file referenced by the FILE% parameter. If the requested lock for the data record GETKEY found cannot be granted, LOKCOD returns a nonzero value. See the SETLOK function for a description of lock codes.
KEYVAL$	The actual key value of the index file record being sought.

Additional Comments:

If your application program takes advantage of the four-byte capacity of the associated data record numbers, your program must call the DATVAL function immediately after GETKEY to return the value of the two high-order bytes. GETKEY returns the value of the two low-order bytes.

Error Codes:

Table 3-17 lists the error codes for the GETKEY Function.

Table 3-17. GETKEY Error Codes

Value	Code	Explanation
30	AN	Index file number (KEY%) is out of range.
46	BN	Index file is not open.
147	IC	Lock code (DLOCK%) is out of range.
153	II	Bad parameter value.

Example:

In this example, notice GETKEY only finds exact matches. Also, CONV.INT converts the part numbers to numeric information.

```
DEF CONV.INT(INTVAL%)
    STRING CONV.INT
    INTEGER MSB,LSB

    MSB = INT%(INTVAL% / 256)
    LSB = INTVAL% - MSB * 256
    IF LSB < 0 THEN MSB = MSB - 1
    CONV.INT = CHR$(LSB) + CHR$(MSB)
FEND

INPUT "Enter Desired Part #:";PART.NO%
PART.NO$ = CONV.INT(PART.NO%)
LOCK% = 1           REM  Shared data record lock
POINTER% = GETKEY(PART.KEY%,INV.FIL%,LOCK%,PART.NO$)
SEGMENT% = DATVAL
IF ERRCOD <> 0 THEN \
   CALL EXCEPTION(2,1)
IF POINTER% = 0 AND SEGMENT% = 0 THEN \
   PRINT "Requested Part Number not in data base"
IF LOCK.CODE% <> 0 THEN \
   PRINT "Shared Record Lock not granted."
```

INTUSR Function

Syntax: INTUSR (PROGID%, ERROPT%, TIMOUT%)

System Initialization and Maintenance Function

Explanation:

INTUSR initializes Access Manager and must be called by your application program prior to calling any other functions.

Parameters:

PROGID% [MULTI] This parameter is ignored in a single-user environment. It should contain the identification number assigned to your application program. Each application program calling the background server must have a unique PROGID% number. PROGID% numbers begin at zero. For example, in a three-user environment the acceptable values for PROGID% are 0, 1, and 2. (See your Programmer's Guide for additional information on system configuration.)

To simplify program development, if PROGID% is set to -1, INTUSR returns the number of the console requesting the application program. This makes it possible to run an application program from different consoles without any changes to set up a unique program ID or having to predetermine the PROGID% values.

ERROPT% A value to indicate how you want Access Manager to handle user errors when they occur.

Set ERROPT% to zero if you want Access Manager to display a two-character error message on the console and then return control to the operating system.

Set ERROPT% to a nonzero value if you want Access Manager to trap user errors and return them to your program for processing. In this case, the ERRCOD function detects user errors, making it possible for your application program to determine the reason for the error and take appropriate action. The value of ERRCOD should be examined following each call to an Access Manager function. Section 4 contains a list of possible ERRCOD values.

TIMOUT% [MULTI] This parameter is ignored in a single-user environment. Your program can use it to specify the number of seconds within which the background server must respond to the INTUSR call. If the background server fails to respond within this timeframe, the following message appears on the console:

Be sure Access Manager is operational. Run MPMSTAT to see.

Control then returns to your operating system.

If TIMOUT% is zero, no response time check is made. Therefore, if Access Manager is not running, the application program hangs indefinitely at the INTUSR call. A reasonable value for TIMOUT% is three seconds. The time check starts only after the call to INTUSR is sent to the background server. If the call must wait in a queue, the time check is not affected.

Additional Comments:

[SINGLE] Must be called at least once, but can be called as often as necessary to change the error handling technique.

[MULTI] Subsequent calls of the INTUSR function clear the message queue and remove all pending locks associated with PROGID%. This can produce unpredictable results and should be avoided.

Error Codes:

[SINGLE] Calls to the INTUSR function should never cause user errors.

[MULTI] User errors might result if PROGID% is outside the allowed range of the background server, or the server is not active when INTUSR is called. The possible errors are listed in Table 3-18.

Table 3-18. INTUSR Error Codes

Value	Code	Explanation
161	JA	Program number (PROGID%) is out of range.
162	JB	Cannot open Access Manager input queue.
163	JC	Cannot open output queue.

Example:

```
PROGID% = INTUSR(-1,1,3)
IF ERRCOD <> 0 THEN \
     PRINT "Cannot Initialize User...Error:";ERRCOD
IF SETUP(7,3,4,1) <> 0 THEN \
     PRINT "Illegal SETUP parameters"
```

[SINGLE] The above statements imply there are seven buffers, up to three index files can be open at one time, the index file record length is 512 bytes (four 128-byte sectors), and only one data file is opened at a time. Further, it is implied that user errors are trapped by calling the ERRCOD function following all calls to Access Manager functions, although calls to INTUSR should never result in user errors.

[MULTI] Note the call to SETUP is ignored. An application program cannot affect the SETUP parameters. These are established at the time the background server is configured.

Error trapping is enabled by the nonzero ERROPT% parameter. Because the PROGID% parameter is passed as -1, INTUSR returns the value corresponding to the console number from which the application program is run. Therefore, the program uses PROGID% to identify the console. More importantly, the same application program can be executed from different consoles without predetermined PROGID% values.

If the assigned PROGID% is beyond the permissible range, INTUSR returns a user error code. But, if error trapping is not enabled, the following message is displayed:

Is PROG ID out of range?

If the background server has not been initiated, the TIMOUT% value of three causes the Run MPMSTAT... message to display (see TIMOUT% parameter description).

If an Access Manager Resident System Process is required but not included at GENSYS, INTUSR returns a user error code. If error trapping is not enabled, the following message is displayed:

Has an Access Manager RSP been included at GENSYS?

LASKEY Function

Syntax: LASKEY (KEY%, FILE%, DLOCK%, IDXVAL$)

Index File Search Function

Explanation:

LASKEY returns the data record number assigned to the last entry in an index file. LASKEY also places the value of the last key it locates in the IDXVAL$ parameter.

Parameters:

KEY% The number of the index file in which you want to find the last key.

FILE% [MULTI] This parameter is ignored in a single-user environment. It is the number of the data file referenced by the KEY% parameter.

DLOCK% [MULTI] This parameter is ignored in single-user environments. It specifies the type of data record lock requested for the data file referenced by the FILE% parameter. If the requested lock for the data record LASKEY found cannot be granted, LOKCOD returns a nonzero value. See the SETLOK function for a description of lock codes.

IDXVAL$ This is an OUTPUT parameter. Access Manager places the last key value found in the index file in IDXVAL$ at the conclusion of the function. If the index is empty, LASKEY returns a zero and IDXVAL$ is set to all blanks (which is not the same as a null string).

Before calling the LASKEY function, IDXVAL$ must be at least as long as the key length specified for the KEY% file or user error 35/BC results. (See KEYLEN% parameter under OPNIDX function.)

Additional Comments:

To determine the two high-order bytes of the associated data record number, call the DATVAL function immediately after LASKEY.

Error Codes:

Table 3-19 lists the error codes for the LASKEY Function.

Table 3-19. LASKEY Error Codes

Value	Code	Explanation
30	AN	Index file number (KEY%) is out of range.
35	BC	IDXVAL$ too short to contain key value.
46	BN	Index file is not open.
147	IC	Lock code (DLOCK%) is out of range.
153	II	Bad parameter value.

Example:

The following example prints data record numbers in reverse key order.

```
REC.LOK% = 1      REM  Shared Record Lock

DRN% = LASKEY(KEY.NO%,FILE.NO%,REC.LOK%,LAST.KEY$)
WHILE DRN% <> 0 AND ERRCOD <> 0 AND LOKCOD <> 0
  PRINT LAST.KEY$,DRN%
  TARGET$ = LEFT$(LAST.KEY$,KEY.LEN%)
  DRN% = BEFKEY(KEY.NO%,FILE.NO%,REC.LOK%,TARGET$,LAST.KEY$)
WEND

IF ERRCOD <> 0 THEN CALL ERROR.HANDLER
IF LOKCOD <> 0 THEN CALL LOCK.CONFLICT
PRINT "The End"
```

LOKCOD Function

Syntax: LOKCOD

System Initialization and Maintenance Function

Explanation:

LOKCOD returns a value indicating whether or not a request to lock a data file or data record was granted.

[SINGLE] Because data file and record locks are unnecessary in a single-user environment, all calls to LOKCOD return zero, indicating a successful lock operation.

When a lock/unlock request is successful, LOKCOD returns a value of zero; otherwise, it returns a nonzero value.

Parameters:

The LOKCOD function is called without parameters.

Additional Comments:

When a data record lock is not granted, LOKCOD returns a one to signify a conflict with another record lock, or a two to signify the data file (not just an individual record) is locked exclusively.

When a request for an exclusive data file lock is not granted, LOKCOD returns a two if another user already holds the exclusive file lock, or a one if another user holds a data record lock in the specified data file.

A request to unlock a data file or individual data record fails only if the expected lock is not found. In this case, LOKCOD returns a one.

Error Codes:

Calls to LOKCOD do not cause user errors.

NEWREC Function

Syntax: NEWREC (FILE%, DLOCK%)

Data File Update Function

Explanation:

NEWREC returns the number of the next available record in a data file.

Parameters:

FILE% The number of the data file where you want to locate the next available data record.

DLOCK% [MULTI] Specifies the type of data record lock operation requested for the data file to be accessed.

If another user holds an exclusive file lock on the data file specified by the FILE% parameter, LOKCOD returns a two and the value returned by NEWREC has no meaning. If you have already opened a data file with either a shared or exclusive file lock, another program cannot get an exclusive file lock. Thus, once you open a data file, you should not run into a lock conflict when calling NEWREC. LOKCOD should never equal one following a call to NEWREC.

Even though you do not normally need to test the LOKCOD function following a call to NEWREC, it can be an aid to debugging your program. See the SETLOK function for a description of lock codes.

Additional Comments:

NEWREC automatically reclaims previously deleted data records (see RETREC function) before extending the size of the data file. That is, if any deleted data records are available, NEWREC uses them first. If not, NEWREC increases the size of the data file to generate a new data record.

To determine the two high-order bytes of the next available data record number, call the DATVAL function immediately after NEWREC.

NEWREC must be called each time a new data record is required. If it is not, Access Manager does not track the actual size of the data file. The next time you attempt to use the RETREC, READAT, or WRTDAT function, user error 52/CD might occur (indicating the data record number is beyond the end of the file).

The first available data record number NEWREC returns for a newly created data file is the first data record beginning on or after the 129th byte of the data file. For example, if the data file record length is greater than or equal to 128 bytes, the first available data record number is two. If the record length is less than 128, the first available record number ranges from three to thirty-three, depending on the record length. (For further information, refer to the OPNDAT function.)

Error Codes:

Table 3-20 lists the error codes for the NEWREC Function.

Table 3-20. NEWREC Error Codes

Value	Code	Explanation
60	CL	Data file number (FILE%) is out of range.
69	DE	First byte of deleted data record not FFH.
177	KA	Lock code (DLOCK%) is out of range.
183	KG	Bad parameter value.

Example:

For this example, assume the data file has been opened successfully.

```
DLOCK% = 2      REM  Exclusive Record Lock Request
DRN% = NEWREC(FILE.NO%,DLOCK%)
IF ERRCOD <> 0 THEN \
     CALL ERROR.HANDLER(3)
IF LOKCOD <> 0 THEN \
     CALL LOCK.CONFLICT(3)
```

NMNODS Function

Syntax: NMNODS (KEY%)

Index File Setup and Maintenance Function

Explanation:

NMNODS returns a count of the number of used and usable records (that is, B-Tree nodes) in an index file. The number will not be more than two bytes in length.

Parameters:

KEY% The number of the index file where the used and usable records are counted.

Additional Comments:

The count returned by the NMNODS function does not include the index file header record.

The NMNODS and GETFDS functions provide a way to compute how much disk space your Access Manager files occupy. (The example at the end of this section shows how to do this for an index file.)

Error Codes:

The NMNODS% function causes these error codes:

Table 3-21. NMNODS Error Codes

Value	Code	Explanation
30	AN	Index file number (KEY%) is out of range.
46	BN	Index file is not open.
153	II	Bad parameter value.

Example:

This example shows how to compute the number of kilobytes occupied by an index file. Note that one is added to TOTAL.NODS to account for the index file header record.

```
TOTAL.NODES = NMNODS(PART.KEY%)
IF TOTAL.NODES < 0 THEN \
   TOTAL.NODES = TOTAL.NODES + 65536.
KILOBYTES% = INT((TOTAL.NODES +1.) * NO.NODES.SECTORS +7 \
   / 8)
```

NOKEYS Function

Syntax: NOKEYS (KEY%)

Index File Setup and Maintenance Function

Explanation:

NOKEYS returns a count of the number of entries in an index file.

Parameters:

KEY% The number of the index file where the entries are counted.

Additional Comments:

NOKEYS returns the two low-order bytes of the number of entries in the index file specified by KEY%. If your application program requires the two high-order bytes, call the DATVAL function immediately after NOKEYS.

Error Codes:

Table 3-22 lists the error codes for the NOKEYS% Function.

Table 3-22. NOKEYS Error Codes

Value	Code	Explanation
30	AN	Index file number (KEY%) is out of range.
46	BN	Index file is not open.
153	II	Bad parameter value.

Example:

Assume you have an inventory index file with more than 65,535 entries. This example might be used to count the number of entries or "parts" in that index file.

```
NO.OF.PARTS = NOKEYS(PART.KEY%)
IF NO.OF.PARTS < 0.0 THEN \
   NO.OF.PARTS = NO.OF.PARTS + 65536.
NO.OF.PARTS = NO.OF.PARTS + (65536. * DATVAL)
```

NXTKEY Function

Syntax: NXTKEY (KEY%, FILE%, DLOCK%, IDXVAL$)

Index File Search Function

Explanation:

NXTKEY returns the data record number associated with the next entry in an index file. NXTKEY also places the key value it finds in the IDXVAL$ parameter.

[SINGLE] NXTKEY is an efficient way to move through an index file sequentially when no updates are performed. You can interleave calls to NXTKEY for different KEY%s because separate position pointers are maintained for each index file.

[MULTI] NXTKEY is not normally used in the multiuser environment.

Parameters:

KEY% The number of the index file where the search takes place.

FILE% [MULTI] This parameter is ignored in a single-user environment. It is the number of the data file referenced by the KEY% parameter.

DLOCK% [MULTI] This parameter is ignored in single-user environments. It specifies the type of data record lock requested for the data file referenced by the FILE% parameter. If the requested lock for the data record that NXTKEY found cannot be granted, LOKCOD returns a nonzero value. See the SETLOK function for a description of lock codes.

IDXVAL$ This is an OUTPUT parameter. Access Manager places the key value found in the index file in IDXVAL$. If no next index entry is found, NXTKEY returns a zero, and IDXVAL$ is set to all blanks (which is not the same as a null string).

Before calling the NXTKEY function, IDXVAL$ must be at least as long as the key length specified for the KEY% file. Normally, it is only necessary to initialize IDXVAL$ once at the beginning of a program.

Additional Comments:

To determine the two high-order bytes of the associated data record number, call the DATVAL function immediately after NXTKEY.

Before the very first call to NXTKEY for a given KEY% or after the index file associated with KEY% is updated, one of the other index search functions, except PRVKEY, must be called for the same KEY%. The other search functions establish the internal pointer used by NXTKEY and PRVKEY for sequential processing. Each time any of the search functions are called, including NXTKEY and PRVKEY, the internal pointer is appropriately updated. The internal pointers are not maintained when the index file is updated.

Error Codes:

Table 3-23 lists the NXTKEY Function error codes.

Table 3-23. NXTKEY Error Codes

Value	Code	Explanation
30	AN	Index file number (KEY%) is out of range.
35	BC	IDXVAL$ too short to contain key value.
46	BN	Index file is not open.
147	IC	Lock code (DLOCK%) is out of range.
153	II	Bad parameter value.

Example:

In the following example, SERKEY finds the first potential match for a customer name, and NXTKEY moves through the index file sequentially to scan the remaining potential matches. This example is only for single-user environments because NXTKEY is used.

The example listing assumes customer names are a maximum of sixteen bytes, and the names entered into the index file are truncated to nine bytes and suffixed with a two-byte, unique sequence number by Access Manager. The data file record length is 64 bytes.

```
DEF SPACE(LENGTH)
    INTEGER LENGTH
    STRING SPACE

REM  12345678901234567890123456789012345678901234567890123456789012345678
SPACE = LEFT$( \
    "                                                                "\
    ,LENGTH)
    RETURN
FEND

DLOCK% = 0
IO.BUFFER$ = SPACE(48) + SPACE(16)
BUFFER.PTR% = SADD(IO.BUFFER$) + 2
ENTRY$ = SPACE(11)

INPUT "Enter Customer Last Name:";CUST.NAME$
CUST.NAME$ = LEFT$(CUST.NAME$ + SPACE(16),16)

REM
REM Target is padded with CHR$ (0) so Access Manager does
REM not pad with blanks. Note these last two bytes can
REM have any binary value because they serve simply as tie-
REM breakers. If Access Manager were allowed to pad with
REM blanks (20H), some key values which matched in the
REM significant bytes (for example, the first nine) might
REM be skipped when SERKEY is called because their last
REM two bytes were less than 2020H.
REM

TARGET$ = LEFT$(CUST.NAME$,9) + CHR$(0) + CHR$(0)

DRN% = SERKEY(CUST.KEY%,FILE%,DLOCK%,TARGET$,ENTRY$)
WHILE LEFT$(TARGET$,9) = LEFT$(ENTRY$,9) AND DRN% <> 0
    IF READAT(FILE%,DRN%,BUFFER.PTR%) <> 0 THEN \
        PRINT "Read Error ";ERRCOD :\
        STOP

REM
REM Assume Customer Name occupies the first sixteen bytes
REM of the data record
REM

    IF LEFT$(IO.BUFFER$,16) = CUST.NAME$ THEN \
        PRINT DRN%;IO.BUFFER$
    DRN% = NXTKEY(CUST.KEY%,FILE%,DLOCK%,ENTRY$)
WEND

PRINT "Search for Customer Name ended"
```

Listing 3-2. NXTKEY Function Program Code

OPNDAT Function

Syntax: OPNDAT (FILE%, DLOCK%, FILNAME$, RECLEN%)

Data File Setup and Maintenance Function

Explanation:

OPNDAT opens a data file. A data file must be opened before it can be read, written, or erased.

At the conclusion of the function, OPNDAT returns the number of the opened data file.

Parameters:

FILE% The number of the data file to be opened. The file number can be an integer between zero and NDATF%-1 (NDATF% is a parameter of the SETUP function). The maximum value of NDATF% depends on your operating system and whether yours is a single-user or multiuser environment. Consult your Programmer's Guide.

Note that file numbers used by Access Manager are separate from those used with your application language. Assigning a particular number to an Access Manager data file does not preclude your using the same file number with an application language file.

If the FILE% parameter contains a value of -1, OPNDAT returns an integer value as follows:

- If an exact match for FILNAME$ (see below) is found among the currently opened data files, OPNDAT returns the number assigned to that file.
- If no match exists for FILNAME$, OPNDAT returns the first file number not in use.
- If no file number is available, OPNDAT causes user error 60/CL, indicating the file number is out of range.

[MULTI] Passing a FILE% parameter of -1 to OPNDAT allows different applications, or different users of the same application, to share the same data files without prior knowledge of file number assignments. If a file is already open and your application program does not use a FILE% parameter of - 1, Access Manager returns user error 63/CO (indicating the file is already in use) or causes a lock conflict.

DLOCK% [MULTI] Specifies the type of lock requested for the data file to be opened.

Set DLOCK% to three to request a shared data file lock. If granted, this lock stops any other user from gaining an exclusive file lock. Any number of users can hold shared file locks simultaneously.

Set DLOCK% to four to request an exclusive file lock. The lock is only granted if no other locks (file or record) are held on the data file by other programs.

If you hold a shared lock on a data file, you can only change to an exclusive file or record lock if no other users hold shared locks at the time. Use the SETLOK function to attempt such a change.

DLOCK% values of one or two should not be used with the OPNDAT function because they refer to individual record locks.

FILNAME$ A character string to indicate the name of the data file you want to open. Access Manager always converts the string value to upper-case letters. The data filename can include a password. A password is designated by a file specification (which includes a semi-colon) followed by the password.

RECLEN% A value to specify the length of the individual records in the data file. Records must be a minimum of four bytes in length.

Additional Comments:

You cannot open a corrupted data file with the OPNDAT function, user error 53/CE results. See the OPRDAT function for an explanation of opening corrupted data files.

Access Manager will assign a password to a data file created with the OPNDAT function if a password is part of the FILNAME$ parameter (for example, B:CUSTOMER.DAT;SECRET where the password is SECRET), and the drive on which the file is created has been set for

automatic XFCB (extended File Control Block) creation. When both conditions are satisfied, the files created by OPNDAT are protected at the highest level; namely, READ protected (the password is required even to read the file). See your operating system documentation to determine if the SET command is available to enable automatic XFCB creation.

[MULTI] Even if automatic XFCB creation is not enabled in a multiuser environment, or if the file was not created with a password, the Access Manager background server will maintain temporary passwords as long as the data file is open. However, the password will not be permanently recorded with the data file.

The first 128 bytes of a data file are reserved for status information. No other data can be stored in this area. Therefore, the first data record available to an application program use is the one beginning at or beyond the 129th byte of the data file. The first available data record number in a file can be determined by the following expression:

```
INT((128 + RECLEN% -1) / RECLEN%) + 1
```

where INT truncates its argument. For example, if the record length is 32 bytes, the first record available for data is number 5. Figure 3-2 shows how the beginning of a data file is organized when the record length is 32 bytes.

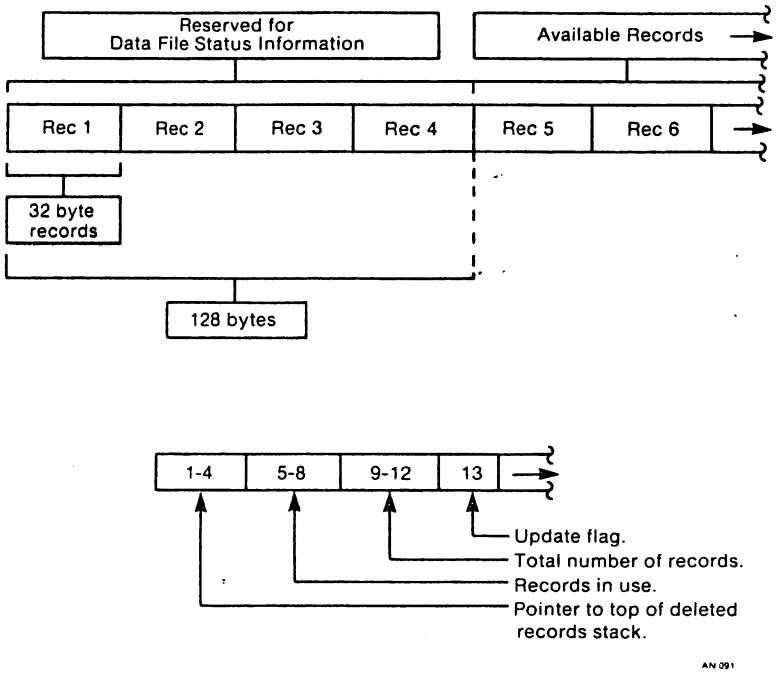

Figure 3-2. Sample Data File Layout for 32-byte Records

Note that the NEWREC function automatically determines the number of the first usable data record; your application program need not compute this number.

Error Codes:

OPNDAT causes the following user errors:

Table 3-24. OPNDAT Error Codes

Value	Code	Explanation
53	CE	Data file is corrupt; no header record.
54	CD	No more directory space.
60	CL	Data file number (FILE%) is out of range.
61	CM	Data filename (FILNAME$) incorrectly formed.
63	CO	Data file (FILE%) is already in use.
70	DF	The data file is corrupt.
119	GG	Incorrect or missing password.
177	KA	Lock code (DLOCK%) is out of range.
183	KG	Bad parameter value.

Example:

```
FILE.NO% = -1    REM  Automatic file number assignment
DLOCK% = 3       REM  Shared file lock
RECORD.LEN% = 100
FILE.NAME$ = "E:CUSTOMER.DAT"
FILE.NO% = OPNDAT(FILE.NO%,DLOCK%,FILE.NAME$,RECORD.LEN%)
IF ERRCOD <> 0 THEN \
        CALL ERROR.HANDLER(1)
IF LOKCOD <> 0 THEN \
        CALL LOCK.CONFLICT(1)
```

Note that the OPNDAT function assigns FILE.NO% a value. If a user error occurs when opening the data file, the program calls an error handling subroutine with a parameter indicating where the error occurred. The error handling routine can then call the ERRCOD function to get the actual error code value. Likewise, if Access Manager refuses the requested data file lock, control transfers to the lock conflict handler.

OPNIDX Function

Syntax: OPNIDX (KEY%, IDXNAME$, KEYLEN%, KEYTYP%, DUPKEY%)

Index File Setup and Maintenance Function

Explanation:

OPNIDX opens or creates an index file and, optionally, assigns the file a number. When a file number is assigned, all subsequent references to the index file are made using this number.

Parameters:

KEY% The number of the index file to be opened or created. The number must range from zero to NKEYS%-1 (see the SETUP function for an explanation of NKEYS%). If KEY% passes -1 to OPNIDX, an integer value is returned as follows:

- If an exact match for the IDXNAME$ parameter (see below) is found among the currently open index files, OPNIDX returns the index file number.
- If no match is found, OPNIDX returns the first unused index file number.
- If no index file number is available, user error 30/AN is returned, indicating KEY% is out of range.

Index file numbers are independent of data file numbers assigned by Access Manager or the application language.

[MULTI] The ability to automatically assign KEY% parameters means application programs running simultaneously can share index files without any concern about how KEY% numbers are assigned.

IDXNAME$ A character string to indicate the name of the index file you want to open or create. Access Manager converts the string to upper-case letters. The index filename might include a password. A password is designated by a file specification (which includes a semi-colon) followed by the password.

KEYLEN% A value specifying the maximum length (in bytes) of the key values in this index file. The value must be at least one and not greater than forty-eight. Integer keys must be at least two bytes in length.

KEYTYP% A value indicating whether the keys in this index file are stored in alphanumerically increasing order or numeric order. Zero indicates alphanumeric keys; one indicates signed, integer keys. See the description of the ADDKEY function for additional information on coding integer key values.

All key values are passed to and from Access Manager as string-valued quantities, even if the key is designated as numeric. The KEYTYP% parameter affects only the order in which keys are stored.

Numeric key values are integers (not necessarily restricted to two bytes) stored with the least-significant byte first and the most-significant, including the sign of the integer, last. Negative integers are stored in two's complement form.

DUPKEY% A value indicating whether or not duplicate key values should be handled by Access Manager. If DUPKEY% is one and KEYTYP% is zero (alphanumeric key values), Access Manager automatically assigns sequence numbers to the last two bytes of each key value. For example, if KEYLEN% is ten, bytes nine and ten of each key value are filled (or overwritten) by Access Manager with a sequence number guaranteed to be unique. These sequence numbers are in binary form and usually do not represent valid ASCII symbols.

DUPKEY% has no effect if KEYTYP% is other than zero.

Note: please read the DELKEY function description for important information concerning the adverse effect of automatic duplicate keys on the time required to delete them.

Additional Comments:

A corrupted index file cannot be opened using the OPNIDX function. Refer to the OPRIDX function for an explanation of opening corrupted index files.

Access Manager assigns a password to an index file created by OPNIDX if a password is part of the IDXNAME$ parameter (for example, CUSTOMER.IDX;SCRAMBLE where the password is SCRAMBLE), and the drive on which the file is created is set for automatic XFCB (extended File Control Block) creation. When both conditions are satisfied, the files created by OPNIDX are protected at the highest level; namely READ protected (the password is required even to read the file). See your operating system documentation to determine if the SET command is available to enable automatic XFCB creation.

[MULTI] Even if automatic XFCB creation is not enabled in a multiuser environment, or if the file was not created with a password, the Access Manager background server will maintain temporary passwords as long as the index file is open. However, the password will not be permanently recorded with the index file.

Based on the index file record length (refer to the NNSEC% parameter under the SETUP function), Access Manager automatically determines the maximum number of key values stored in each index file record. As the number of key values per index file record increases, the number of levels in the index structure decreases. The number of levels in the index structure determines the maximum number of disk accesses required to locate a key value. Of course, if an index file record is in an Access Manager I/O buffer, no disk access is required. Also, if the index file record length is greater than the physical sector size of the disk, each logical access can require two or more disk accesses.

The maximum key values per index file record is computed as the largest even integer less than or equal to

```
((NNSEC% * 128) - 10) / (KEYLEN% + 4)
```

Further, the maximum number must be at least four, or user error 39/BG occurs. Access Manager restricts the maximum number of key values per index file record. These restrictions are stated in your Programmer's Guide.

Once you choose values for NNSEC% and KEYLEN%, you can compute MAXKV (the maximum number of key values per index file record). For a computed MAXKV, the minimum possible index file size (in bytes) for a given number of index entries is computed in Pascal/MT+ as follows:

```
NODES = <ENTRIES / MAXKV> [where <X> is the smallest
                    integer greater than or equal to X]
INDEX_NODES = NODES
WHILE INDEX_NODES > 1
     INDEX_NODES = <INDEX_NODES / (MAXKV + 1)>
     NODES = NODES + INDEX_NODES
WEND
INDEX_FILE_SIZE = (NODES + 1) * NNSEC% * 128
```

To compute the largest possible index file size, replace MAXKV by MAXKV/2 in the preceding algorithm. The minimum size computation assumes completely full nodes, while the largest size computation assumes half-full nodes. The ordinary B-Tree structure ensures at least half-full nodes. Access Manager performs local node rotations that help maintain the smallest index file size by avoiding unnecessary node splitting.

For example, assume NNSEC% is four and KEYLEN% is ten. Access Manager computes MAXKV to be thirty-four. The minimum bytes required to index 10,000 key values under these circumstances is 156,672 (153K). The maximum number of bytes required is 320,512 (313K). Actual experience with Access Manager indicates for random insertions the index file records are approximately three- quarters full. This corresponds to an estimated file size of 210,432 bytes (206K).

Because user errors are automatically sent to the console in this example listing (ERROPT%, the second parameter of INTUSR, equals 0), there are no tests of the ERRCOD function.

```
REM ------------------------------------------------
REM   AM80 External Declarations
REM ------------------------------------------------

%INCLUDE AM80EXTR.BAS

REM ------------------------------------------------
REM   Parameter Setup
REM ------------------------------------------------

YES% = 1
NO% = 0
PROGID% = -1
ERROR.TRAP% = NO%
TIMOUT% = 0
BUFS% = 10
KEYS% = 2
NSEC% = 4
DFILE% = 1
```

Listing 3-3. OPNIDX Function Program Code

```
REM ------------------------------------------------
REM   Initialize AM80
REM ------------------------------------------------

PROGID% = INTUSR%(PROGID%,ERROR.TRAP%,TIMOUT%)
DUMMY% = SETUP(BUFS%,KEYS%,NSEC%,DFILE%)

REM ------------------------------------------------
REM   Open Index Files
REM ------------------------------------------------

CUST.IDX$ = "B:CUST.IDX"
PART.IDX$ = "C:PART.IDX"
CUST.TYPE% = 0            REM  Alphanumeric Key
PART.TYPE% = 1            REM  Integer Key
CUST.LEN% = 11
PART.LEN% = 4
CUST.DUP% = YES%
PART.DUP% = NO%    REM  No duplicate part numbers

CUST.KEY% = OPNIDX(-1,CUST.IDX$,CUST.LEN%,CUST.TYPE%,CUST.DUP%)
PART.KEY% = OPNIDX(-1,PART.IDX$,PART.LEN%,PART.TYPE%,PART.DUP%)

REM ------------------------------------------------
REM   Open Data File
REM ------------------------------------------------

INV.DAT$ = "D:INVENTRY.DAT"
INV.LEN% = 192                REM Record length
S.FILE% = 3                   REM Shared file lock
INV.FILE% = OPNDAT(-1,S.FILE%,INV.DAT$,INV.LEN%)
```

Listing 3-3. (continued)

Error Codes:

Table 3-25 lists the OPNIDX Function error codes.

Table 3-25. OPNIDX Error Codes

Value	Code	Explanation
23	AG	Index file is corrupt; no header record.
24	AH	No more directory space.
30	AN	Index file number (KEY%) is out of range.
31	AO	Illegal index filename.
33	AQ	Index file number (KEY%) is already in use.
34	BB	Key length (KEYLEN%) exceeds 48 bytes.
39	BG	Key length (KEYLEN%) too long for number of assigned disk sectors (NNSEC%).
40	BH	Index file is corrupted.
89	EI	Incorrect or missing password.
53	II	Bad parameter value.

OPRDAT Function

Syntax: OPRDAT (FILE%, DLOCK%, FILNAME$, RECLEN%)

Data File Setup and Maintenance Function

Explanation:

OPRDAT is used in place of the OPNDAT function to open a corrupted data file. A data file is corrupted when it has been updated but not subsequently closed by a SAVDAT or CLSDAT function. OPRDAT performs exactly like the OPNDAT function except it does not check to see if the data file is corrupted. OPRDAT can be used to open and reconstruct a corrupted data file.

At the conclusion of the function, the number of the opened data file is returned.

Note: a corrupted data file opened with OPRDAT and then subsequently saved (SAVDAT) or closed (CLSDAT) will no longer appear corrupted to Access Manager even if it still is corrupted. Therefore, use OPRDAT with due caution.

Parameters:

Parameters for OPRDAT are exactly the same as those for OPNDAT. Please refer to OPNDAT for explanations of these parameters.

Error Codes:

OPRDAT causes the following user errors:

Table 3-26. OPRDAT Error Codes

Value	Code	Explanation
53	CE	No header record in data file.
54	CD	No more directory space.
60	CL	Data file number (FILE%) is out of range.
61	CM	Data filename (FILNAME$) incorrectly formed.
63	CO	Data file (FILE%) number already in use.
119	GG	Incorrect or missing password.
177	KA	Lock code (DLOCK%) is out of range
183	KG	Bad parameter value.

Example:

This examples demonstrates how to create a data file that is guaranteed to be new. The old data file is erased first, even if it is corrupted.

```
FILE.NO% = -1
DLOCK% = 4          REM  Exclusive file lock
RECORD.LEN% = 100
FILE.NAME$ = "E:DUMMY.DAT"

FILE.NO% = OPRDAT(FILE.NO%,DLOCK%,FILE.NAME$,RECORD.LEN%)
IF ERRCOD <> 0 AND ERRCOD <> 53 THEN \
   CALL ERROR.HANDLER(1)
IF LOKCOD <> 0 THEN CALL LOCK.CONFLICT(1)

REM  If no errors, erase old file and create new, empty file.

DUMMY% = ERADAT(FILE.NO%,DLOCK%)
FILE.NO% = OPNDAT(-1,DLOCK%,FILE.NAME$,RECORD.LEN%)
IF ERRCOD <> 0 THEN CALL ERROR.HANDLER(2)
IF LOKCOD <> 0 THEN CALL LOCK.CONFLICT(2)
```

OPRIDX Function

Syntax: OPRIDX (KEY%, IDXNAME$, KEYLEN%, KEYTYP%, DUPKEY%)

Index File Setup and Maintenance Function

Explanation:

OPRIDX opens or creates an index file that cannot be opened or created using the OPNIDX function.

Each time OPNIDX opens an index file, it checks the integrity of the file. Integrity is lost if the file has been updated but not subsequently closed using the SAVIDX or CLSIDX functions. If OPNIDX discovers a loss of integrity in the index file, the file is not opened, instead, user error 40/BH is issued.

OPRIDX behaves exactly the same as the OPNIDX function except it does not check the integrity of the index file.

Parameters:

Parameters for OPRIDX are exactly the same as for the OPNIDX function. Please refer to the OPNIDX Function for a discussion of these parameters.

Additional Comments:

OPRIDX is typically used with the ERAIDX function. Any other use of OPRIDX can cause unexpected error conditions, including Access Manager internal consistency errors.

An index file opened with OPRIDX and subsequently closed using SAVIDX or CLSIDX might appear uncorrupted to Access Manager when in fact it is corrupted. If an index must be opened with OPRIDX, replace or rebuild it immediately.

Error Codes:

OPRIDX causes the error codes listed in Table 3-27.

Table 3-27. OPRIDX Error Codes

Value	Code	Explanation
23	AG	No header record in index file.
24	AH	No more directory space.
30	AN	Index file number (KEY%) is out of range.
31	AO	Illegal index filename.
33	AQ	Index file number (KEY%) is already in use.
34	BB	Key length (KEYLEN%) exceeds 48 bytes.
39	BG	Key length (KEYLEN%) is too long for the number of assigned disk sectors (NNSEC%).
153	II	Bad parameter value.

Example:

This example demonstrates how to create an index file guaranteed to be new and empty (possibly for temporary use). Note that OPRIDX should be used in conjunction with the ERAIDX and OPNIDX functions.

```
KEY% = -1
KEY.NAME$ = "DUMMY.IDX"
KEY.LEN% = 10
KEY.TYP% = 0
KEY.DUP% = 0

KEY% = OPRIDX(KEY%,KEY.NAME$,KEY.LEN%,KEY.TYP%,KEY.DUP%)
IF ERRCOD <> 0 AND ERRCOD <> 23 THEN \
   CALL ERROR.HANDLER(1)

DUMMY% = ERAIDX(KEY%)
KEY% = OPNIDX(-1,KEY.NAME$,KEY.LEN%,KEY.TYP%,KEY.DUP%)
IF ERRCOD <> 0 THEN \
   CALL ERROR.HANDLER(2)
```

PRVKEY Function

Syntax: PRVKEY (KEY%, FILE%, DLOCK%, IDXVAL$)

Index File Search Function

Explanation:

PRVKEY returns the data record number associated with the preceding key value in an index file. PRVKEY also places the value of the key it finds in the IDXVAL$ parameter.

PRVKEY is an efficient way to move sequentially backward through an index file in a single-user environment when no index file updates are performed. You can interleave calls to PRVKEY for different KEY%s because separate position pointers are maintained for each index file.

[MULTI] PRVKEY is not normally used in a multiuser environment.

Parameters:

KEY% The number of the index file where the search takes place.

FILE% [MULTI] This parameter is ignored in a single-user environment. It is the number of the data file referenced by the KEY% parameter.

DLOCK% [MULTI] This parameter is ignored in single-user environments. It contains a code to specify the type of data record lock requested for the data file referenced by the FILE% parameter. If the requested lock for the data record PRVKEY found cannot be granted, LOKCOD returns a nonzero value. See the SETLOK function for a list of acceptable lock codes.

IDXVAL$ This is an OUTPUT parameter. Access Manager places the key value found in the index file in IDXVAL$ at the conclusion of the function. If no previous index entry is found, PRVKEY returns a zero, and IDXVAL$ is set to all blanks (which is not the same as a null string).

Before calling the PRVKEY function, IDXVAL$ must be at least as long as the key length specified for the KEY% file. Normally, it is only necessary to initialize IDXVAL$ once at the beginning of a program.

Additional Comments:

To determine the two high-order bytes of the associated data record number, call the DATVAL function immediately after PRVKEY.

Before the very first call to PRVKEY for a given KEY% or after the index file associated with KEY% is updated, one of the other index search functions, except NXTKEY, must be called for the same KEY%. The other search functions establish the internal pointer used by PRVKEY and NXTKEY for sequential processing. Each time any of the search functions are called, including NXTKEY and PRVKEY, the internal pointer is appropriately updated. However, the internal pointers are not maintained when the index file is updated.

Error Codes:

Table 3-28 lists the PRVKEY Function user errors.

Table 3-28. PRVKEY Error Codes

Value	Code	Explanation
30	AN	Index file number (KEY%) is out of range.
35	BC	IDXVAL$ is too short to contain key value.
46	BN	Index file is not open.
147	IC	Lock code (DLOCK%) is out of range.
153	II	Bad parameter value.

READAT Function

Syntax: READAT (FILE%, DRN%, BUFFER%)

Data File Update Function

Explanation:

READAT reads a specified data record into a buffer area in memory. If the read operation is successful, READAT returns a zero at the conclusion of the function; otherwise, a nonzero user error code results.

Parameters:

FILE%	The number of the data file from which the data record is read.
DRN%	The number of the data record to be read from the file specified by the FILE% parameter.
BUFFER%	The address of a buffer area in memory. When the data record is read from the data file, it is placed in this buffer area for subsequent processing.

Additional Comments:

Your application program must ensure that sufficient space is allocated at the location specified by the BUFFER% parameter to accommodate the data records. In other words, make sure your buffer area is large enough to contain your data records.

Your application program must also parse the data record into the desired variables for use. This is easily accomplished in languages permitting based variables and/or data structures.

If the data record number (DRN%) exceeds two bytes, call the SETDAT function immediately before READAT to set the two high-order bytes of the data record number.

Error Codes:

READAT returns zero if the read operation is successful; otherwise, a nonzero value is returned. User errors that might result are listed in Table 3-29.

Table 3-29. READAT Error Codes

Value	Code	Explanation
52	CD	Data record number (DRN%) is zero or beyond the logical end-of-file.
53	CE	Attempt to read past physical end-of-file or read unwritten data.
60	CL	Data file number (FILE%) is out of range.
183	KG	Bad parameter value.

Example:

```
READ.BUFFER$ = SPACE$(RECORD.LENGTH%)
ADR.BUFFER% = SADD(READ.BUFFER$) + 2

IF READAT(FILE.NO%,DRN%,ADR.BUFFER%) <> 0 THEN \
   CALL ERROR.HANDLER

FOR FLD% = 1 TO NO.FLDS%
   FIELD$(FLD%) = MID$(READ.BUFFER$,FIELD.BEG%(FLD%), \
                  FIELD.LEN%(FLD%))
NEXT FLD%
```

RETREC Function

Syntax: RETREC (FILE%, DLOCK%, DRN%)

Data File Update Function

Explanation:

RETREC returns data records to the pool of available records for subsequent reuse with the NEWREC function. If the RETREC function is successful, zero is returned at its conclusion; otherwise, a nonzero user error code results.

Parameters:

FILE% The number of the data file where data records are reclaimed.

DLOCK% [MULTI] This parameter is ignored in a single-user environment. It contains a code specifying the type of data record lock requested for the data file to be accessed. See the SETLOK function for a list of acceptable lock codes.

If Access Manager grants the requested data record lock, RETREC places the returned data record number at the top of a logical stack of deleted records. If a lock conflict occurs, no action is taken and LOKCOD returns the appropriate nonzero value.

Note that unless RETREC is called with a DLOCK% parameter of zero, you must test the return value of LOKCOD immediately after calling RETREC. If LOKCOD returns a nonzero value, the data record is not deleted and your application program must take appropriate action. Further, if RETREC is successful, the data record lock held by the calling program is automatically released.

RETREC automatically releases the data record lock after deleting the data record. This ensure that no other program gets the just deleted record from NEWREC before the deleting program releases its data record lock. In short, the deletion and release-lock operations work as an automatic operation. Therefore, it is unnecessary to call FRELOK after RETREC.

DRN% The data record number of the record to be returned to the pool.

Additional Comments:

If the data record number requires more than two bytes, call the SETDAT function immediately after RETREC.

The stack of deleted records is implemented by a linked list that ensures fast operation because the NEWREC and RETREC functions manipulate only the topmost record in the stack. After a successful call to RETREC, the record in the data file with record number DRN% has two fields written over the previous contents. These fields are defined in the following table:

Table 3-30. Contents of Deleted Data Record

Byte Position	Contents
0	0FFH (255 decimal). This byte serves as a flag for deleted data records. Your program should reserve this byte and ensures it never otherwise contains 0FFH.
1-3	A link to the next available data record.
4+	Undefined.

The 0FFH value in the first byte of deleted data records serves two purposes. First, Access Manager checks the first byte of every deleted record it is about to reuse to see if it contains 0FFH. If it does not, user error 69/DE results. Second, your application program can use the first byte as a deleted record flag. Therefore, when reading through a data file without accessing the index files, you can disregard data records with 0FFH in the first byte. This latter use assumes you either reserved the first byte for the delete flag or your valid data can never contain 0FFH in the first byte.

Error Codes:

Table 3-31 lists the error codes caused by the RETREC Function.

Table 3-31. RETREC Error Codes

Value	Code	Explanation
52	CD	Data record number (DRN%) is zero of beyond the logical end-of-file.
60	CL	Data file number (FILE%) is out of range.
69	DE	First byte of record is not 0FFH.
177	KA	Lock code (DLOCK%) is out of range.
183	KG	Bad parameter value.

Example:

In this example, DRN2% contains the two high-order bytes and DRN% the two low-order bytes of the data record number to be returned to the pool of available data records. Note the use of SETDAT to initialize DRN2%. The arguments of the exception processing functions indicate the location of the exception.

```
DLOCK% = 2      REM  Exclusive Record Lock Request Code
CALL SETDAT(DRN2%)

IF RETREC(FILE.NO%,DLOCK%,DRN%) <> 0 THEN \
   CALL ERROR.HANDLER(4)
IF LOKCOD <> 0 THEN \
   CALL LOCK.CONFLICT(4)
```

SAVDAT Function

Syntax: SAVDAT (FILE%)

Data File Setup and Maintenance Function

Explanation:

SAVDAT forces data file updates to the disk while keeping the data file open for further use. If the save operation is successful, a zero is returned at the conclusion of the function; otherwise, a nonzero user error results.

SAVDAT has the same affect on a data file as issuing a call to CLSDAT followed by a call to OPNDAT, except that if there are no updates, SAVDAT simply returns without performing any actions.

Parameters:

FILE% The number of the saved data file.

Additional Comments:

The main use of the SAVDAT function is to save data file updates at critical points in your application program. Hardware or software failures cannot corrupt a saved data file unless additional updates have been performed subsequent to the save.

If you make changes to a data file, your application program must call the SAVDAT or CLSDAT function to force the updates to the disk file. If you do not do this, the integrity of the data file can be disrupted because the Access Manager header record could be incorrect. Further, the operating system might be holding data in internal buffers. SAVDAT and CLSDAT force the updated header record to the disk and flush the operating system buffers. Calling SAVDAT after each new record is added to a data file will degrade system performance. You must balance the desire for security with the need for quick response times.

Error Codes:

Table 3-32 lists the user errors for the SAVDAT Function.

Table 3-32. SAVDAT Error Codes

Value	Code	Explanation
60	CL	Data file number (FILE%) is out of range.
73	DI	Could not reopen data file during save.
177	KA	Lock code (DLOCK%) is out of range.
183	KG	Bad parameter value.

Example:

```
IF SAVDAT(FILE.NO%) = 0 THEN \
   PRINT "File save was successful."
```

SAVIDX Function

Syntax: SAVIDX (KEY%)

Index File Setup and Maintenance Function

Explanation:

SAVIDX forces index file updates to the disk while keeping the file open for further use. If the save operation is successful, zero is returned at the conclusion of the function; otherwise, a nonzero user error code results.

SAVIDX has the same affect on an index file as issuing a call to CLSIDX followed by a call to OPNIDX, except that if there are no updates, SAVIDX simply returns without performing any actions.

Parameters:

KEY% The number of the saved index file.

Additional Comments:

The main use of the SAVIDX function is to save index file updates at critical points in your application program. Hardware or software failures cannot corrupt a saved index file unless additional updates have been performed since SAVIDX was used.

If you make changes to an index file, your application program must call the SAVIDX or CLSIDX function to force the updates to the disk file. If this is not done, the integrity of the index file is not ensured because some updated nodes might still be in an I/O buffer and/or the header record might be incorrect. Calling SAVIDX after each update of an index file will degrade system performance. You must balance the need for security with that for speedy updates.

Error Codes:

The SAVIDX function causes the following user errors:

Table 3-33. SAVIDX Error Codes

Value	Code	Explanation
21	AE	Disk or directory full.
30	AN	Index file number (KEY%) is out of range.
43	BK	Could not reopen index file during save.
46	BN	Index file (KEY%) is not open.
153	II	Bad parameter value.

SERKEY Function

Syntax: SERKEY (KEY%, FILE%, DLOCK%, KEYVAL$, IDXVAL$)

Index File Search Function

Explanation:

SERKEY returns the data record number assigned to the first entry (in key-sequential order) that is equal to or greater than a specific key value. SERKEY also places the key value it finds in the IDXVAL$ parameter.

SERKEY can be used to locate an entry in an index file when only the beginning portion of the key value is known.

Parameters:

KEY% The number of the index file where the search takes place.

FILE% [MULTI] This parameter is ignored in a single-user environment. It is the number of the data file referenced by the KEY% parameter.

DLOCK% [MULTI] This parameter is ignored in single-user environments. It contains the code specifying the type of data record lock requested for the data file referenced by the FILE% parameter. If the requested lock for the data record SERKEY found cannot be granted, LOKCOD returns a nonzero value. See the SETLOK function for a list of acceptable lock codes.

KEYVAL$ The value of the key record being used as a reference.

IDXVAL$ This is an OUTPUT parameter. Access Manager places the key value found in the index file in IDXVAL$. If no index entry is found with a key value equal to or greater than KEYVAL$, SERKEY returns a zero and IDXVAL$ is set to all blanks (which is not the same as a null string). Also see "Additional Comments" below.

Additional Comments:

If you must determine the two high-order bytes of the data record number found by SERKEY, call the DATVAL function immediately after SERKEY.

There are three additional notes to consider concerning the IDXVAL$ parameter:

- Access Manager never changes the address or length of IDXVAL$ (which is a string-valued parameter). If IDXVAL$ is not initially set to a string value that is long enough to contain the string-valued index entries assigned to IDXVAL$, Access Manager cannot make the assignment (this results in user error 35/BC). At the beginning of an application program, you must set up one or more string variables that can be used in subsequent calls to SERKEY and to the other Access Manager functions that return key values from the index. This can be accomplished by assigning sufficiently long, blank strings to these variables.

- If IDXVAL$ is longer than the KEYLEN% associated with KEY%, IDXVAL$ is padded on the right with blanks. (See OPNIDX function description.)

- KEYVAL$ and IDXVAL$ cannot be the same variable because IDXVAL$ might be initialized to all blanks before the actual search of the index file.

As mentioned under "Duplicate Key Values" in the ADDKEY function, situations can arise where key values must be modified to accommodate duplicate key values. When this is done, you can no longer use GETKEY to locate such an index entry because the sequence number modifies the original value. SERKEY can find the first candidate for a match with the specified key value. Subsequent candidates are found using the NXTKEY and/or AFTKEY function. KEYVAL$ must be set up to avoid automatic padding by Access Manager.

Error Codes:

Table 3-34 lists the SERKEY Function user errors.

Table 3-34. SERKEY Error Codes

Value	Code	Explanation
30	AN	Index file number (KEY%) is out of range.
46	BN	Index file is not open.
147	IC	Lock code (DLOCK%) is out of range.
153	II	Bad parameter value.

Example:

This example shows how to use SERKEY to find the first key value in a set of duplicates. Assume the function SPACE$ has been defined and returns a blank string equal in length to its argument value. Also, assume KEY.LEN% is the key length including the two-byte sequence number automatically set by the ADDKEY function.

```
IDXVAL$ = SPACE$(KEY.LEN%)
INPUT "Enter the target key value: ";KEY.VALUE$

KEY.VALUE$ = LEFT$(KEY.VALUE$ + SPACE$(KEY.LEN%),KEY.LEN%-2)
KEY.VALUE$ = KEY.VALUE$ + CHR$(0)
DRN% = SERKEY(KEY%,FILE%,DLOCK%,KEY.VALUE$,IDXVAL$)
```

SETDAT Function

Syntax: SETDAT (DRN%)

Data File Setup and Maintenance Function

Explanation:

SETDAT passes the two high-order bytes of the four-byte data record number when needed. If you do not call SETDAT, the two high-order bytes are set to zero.

Use SETDAT only when the input data record number parameter (DRN%) exceeds the two-byte capacity of an ordinary integer variable (65,535).

Parameters:

DRN% The two high-order bytes of the data record number.

Additional Comments:

SETDAT does not return a value; it sets the two high-order bytes for data record numbers used as input parameters in other Access Manager functions (such as RETREC).

SETDAT is always used with another Access Manager function and is called immediately before that function.

Error Codes:

SETDAT does not cause user errors.

Example:

```
CALL SETDAT(SEG.NO%)
RET.CODE% = ADDKEY(KEY%,FILE.NO%(1),XCLSV.LOCK%,KEYVAL$,DRN%)

IF ERRCOD <> 0 THEN CALL ERROR.HANDLER
IF LOKCOD <> 0 THEN CALL LOCK.CONFLICT
IF RET.CODE <> 1 THEN CALL ADDKEY.PROBLEM
```

SETLOK Function

Syntax: SETLOK (FILE%, DLOCK%, DRN%)

Data Locking Function

Explanation:

SETLOK sets a lock on a data file or data record. If the lock operation is successful, zero is returned at the conclusion of the function; otherwise, a one or two is returned. This function has no effect in a single-user environment.

Parameters:

FILE%	The number of the data file for which the lock is requested.
DLOCK%	The code to specify the type of lock operation requested. Table 3-35 lists the lock requests that can be requested with this function.
DRN%	The number of the data record for which the lock is requested.

Additional Comments:

If the requested lock is granted, LOKCOD is set to zero and the calling program (see PROGID% parameter under the INTUSR function) holds the lock.

If the requested lock cannot be granted, LOKCOD is set to one or two. One indicates a record lock conflict; two indicates a conflict with an exclusive file lock.

Acceptable lock request codes are shown in Table 3-35.

Table 3-35. Data File/Data Record Lock Requests

DLOCK% Value	Lock Request
0	Locks are ignored but LOKCOD returns zero.
1	Attempt a shared record lock on record number DRN%.
2	Attempt an exclusive record lock on DRN%.
3	Attempt a shared file lock on FILE%.
4	Attempt an exclusive file lock on FILE%.

The ignore lock request (DLOCK% = 0) always returns a successful LOKCOD result. Therefore, unless your program relies on locking protocols outside Access Manager, use caution when passing DLOCK% with a zero. This lock request is most commonly used when your program must search an index file but you do not want to access the data record associated with the returned key value.

Use a shared data record lock (DLOCK% = 1) when you want to review the record but have no intention (at least not yet) of updating it. The shared lock blocks other users from gaining an exclusive lock, but still allows them to access the record for review.

Use an exclusive data record lock (DLOCK% = 2) when you want to update a data record. This prevents two users from updating the same record at the same time.

A user holding a shared record lock can try to change to an exclusive lock; but an attempt to change to an exclusive lock on a data record that already carries shared locks by other users fails with a LOKCOD of one.

A shared file lock (DLOCK% = 3) ensures that the user cannot be subsequently blocked from the file by another user's exclusive file lock.

Use an exclusive data file lock (DLOCK% = 4) when it is imperative that no other users have access to the records in a data file. An exclusive file lock is granted only if no other users have shared locks on the file and there are no active data record locks.

When one user holds an exclusive file lock, all other nonzero lock requests (that is, DLOCK% = 1, 2, 3, or 4) by other users are denied with a LOKCOD of two.

Index file functions can set data record locks at the time the index file operation takes place. Therefore, calling SETLOK when a data record is located by the index file functions is not normally necessary.

Error Codes:

SETLOK Function error codes are listed in Table 3-36.

Table 3-36. SETLOK Error Codes

Value	Code	Explanation
52	CD	Data record number (DRN%) is zero or beyond the logical end-of-file.
60	CL	Data file number (FILE%) is out of range.
177	KA	Lock code (DLOCK%) is out of range.
183	KG	Bad parameter value.

Example:

For this example, assume a shared record lock is held on the data record (DRN%). The purpose is to upgrade the shared lock to an exclusive record lock.

```
IF SETLOK(FILE.NO%,2,DRN%) <> 0 THEN \
   PRINT "Exclusive lock failed."

IF ERRCOD <> 0 THEN CALL ERROR.HANDLER
```

SETUP Function

Syntax: SETUP (NBUFS%, NKEYS%, NNSEC%, NDATF%)

System Initialization and Maintenance Function

Explanation:

SETUP only applies in single-user environments. Calls to SETUP are ignored in a multiuser environment because the background server automatically calls the function.

SETUP prepares the special Access Manager buffer area and specifies the basic characteristics of the index files. You must complete the SETUP function before opening or using any index and/or data files in your program.

Parameters:

NBUFS% A value specifying the number of index file I/O buffers to be used. There must be at least three of these buffers.

As the number of buffers increases, the time to access a key value decreases and the memory space required increases. All index files share the same buffer space, thereby minimizing the memory required in your program. Even if several index files are used simultaneously, it is not necessary to use more than three buffers.

NKEYS% A value specifying the maximum number of index files the program will use simultaneously. The value must be at least one.

NNSEC% Determines or specifies the length of the records in an index file. Specifically, NNSEC% is the number of 128-byte disk sectors in each index file record. Each record corresponds to a B-Tree node. The more sectors per record, the more key values stored per node. The more key values stored per node, the fewer accesses required to find a key value.

NNSEC% must be at least one. There is no upper limit for this parameter, but because there are limits on the number of key values stored per node, you should consider the following practical limitations.

- Access Manager forces all index files open at the same time to have the same record length.

- To maintain compatibility with application software from other vendors, Digital Research suggests a value of four as an informal standard for the NNSEC% parameter. Compatibility is particularly important for multiuser environments because different software can operate simultaneously.

NDATF% A value specifying the maximum number of data files that are open at one time. It is not necessary to use the Access Manager data file functions. You can use routines coded in the application language or in whatever form is appropriate for your programs.

Additional Comments:

See the "Access Manager Design Constraints" section of your Programmer's Guide for additional information on the maximum values permitted for SETUP parameters.

Although unlikely, you might want to change the SETUP parameters during the course of your application program. If you do, be certain to close all index and data files before subsequent calls to SETUP.

Error Codes:

The SETUP function returns a zero if the parameter values fall within their legal ranges and the buffer area is large enough. Error codes that can result are listed in Table 3-37.

Table 3-37. SETUP Error Codes

Value	Code	Explanation
209	MA	Illegal parameter value or buffer area too small.

Example:

See the INTUSR function description.

UPDPTR Function

Syntax: UPDPTR (KEY%, FILE%, DLOCK%, KEYVAL$, DRN%)

Index File Update Function

Explanation:

UPDPTR changes the data record number assigned to an existing key value. At the conclusion of the function, a value is returned to indicate the success or failure of the update operation (see Additional Comments below).

Without UPDPTR, you would first have to delete KEYVAL$ from the index file and then reinsert it with its new data record number (DRN%). UPDPTR provides a more efficient method for doing this.

Parameters:

KEY% The number of the index file where the data record number is changed.

FILE% [MULTI] This parameter is ignored in single-user environments. It contains the number of the data file referenced by the KEY% parameter.

DLOCK% [MULTI] This parameter is ignored in single-user environments. It contains a code specifying the type of data record lock requested for the data file referenced by the FILE% parameter. See the SETLOK function for a list of acceptable lock codes.

To ignore locking protocols in a multiuser environment, assign DLOCK% a value of zero. This procedure, however, is not recommended.

KEYVAL$ The key value for the record where the data record number is changed.

If sequence numbers are assigned for duplicate keys by Access Manager (see ADDKEY function), KEYVAL$ must include the proper sequence number.

DRN% The data record number assigned to the key value contained in KEYVAL$. Specifically, this is the new number assigned to the key value.

Additional Comments:

UPDPTR returns one of the values listed in Table 3-38.

Table 3-38. UPDPTR Function Values

Value	Meaning
0	KEYVAL$ was not found in the index.
1	The data record number was successfully changed.
4	The requested DLOCK% was not granted for the specified data record (DRN%). The change is not made nor is the index file searched for KEYVAL$.

Error Codes:

Table 3-39 lists the user errors for the UPDPTR Function.

Table 3-39. UPDPTR Error Codes

Value	Code	Explanation
30	AN	Index file number (KEY%) is out of range.
46	BN	Index file is not open.
147	IC	Lock code (DLOCK%) is out of range.
153	II	Bad parameter value.

Example:

```
CALL SETDAT(DRN2%)
IF UPDPTR(KEY%,0,0,KEY.VALUE$,DRN%) <> 1 THEN \
   PRINT "Index file pointer not updated."
IF ERRCOD <> 0 THEN \
   CALL ERROR.HANDLER
```

WRTDAT Function

Syntax: WRTDAT (FILE%, DRN%, BUFFER%)

Data File Update Function

Explanation:

WRTDAT writes a data record into a data file. The data record to be written is taken from a buffer area in your computer's memory. If the write operation is successful, a zero is returned at the conclusion of the function; otherwise, a nonzero user error code results.

Parameters:

FILE%	The number of the data file into which the data record is written.
DRN%	The relative number of the data record to be written into the file specified by the FILE% parameter.
BUFFER%	The address of a buffer area from which the data record is written into the data file.

Additional Comments:

Your application program must ensure that sufficient space is allocated at the location specified by the BUFFER% parameter to accommodate the data records. In other words, make sure your buffer area is large enough to contain your data records.

If the data record number (DRN%) exceeds two bytes, call the SETDAT function immediately before WRTDAT to set the two high-order bytes of the data record number.

Error Codes:

If WRTDAT returns a nonzero value, a user error occurred during the write operation. Listed in Table 3-40 are the error codes for the WRTDAT Function.

Table 3-40. WRTDAT Error Codes

Value	Code	Explanation
51	CC	Disk or directory is full.
52	CD	Attempt to write record zero or write past logical end-of-file.
60	CL	Data file number (FILE%) is out of range.
183	KG	Bad parameter value.

Example:

```
WRITE.BUFFER$ = " "
FOR FLD% = 1 TO NO.FLDS%
    WRITE.BUFFER$ = WRITE.BUFFER$ + FIELD$(FLD%)
NEXT FLD%

ADR.BUFFER% = SADD(WRITE.BUFFER$) + 2
IF WRTDAT(FILE.NO%,DRN%,ADR.BUFFER%) <> 0 THEN \
    CALL ERROR.HANDLER(LOCALE%)
```

End of Section 3

Section 4
Access Manager Error Codes

4.1 Error Types

Access Manager functions generate two types of errors: internal consistency errors and user errors.

4.1.1 Internal Consistency Errors

Access Manager generates internal consistency errors when a B-Tree index file or a data file does not satisfy the internal consistency checks the Access Manager functions perform when a file is used. Such errors do not occur unless your application is processing corrupted files or a pointer variable is not properly initialized. If an Access Manager internal consistency error does occur, the console displays an error message of the following form:

```
Access Manager Internal Error ...XY...
```

where XY is replaced by one of the codes listed in Table 4-1.

If, after reviewing your application program you cannot find any obvious cause for the internal error and cannot successfully recreate the file, please contact Digital Research. Provide as much information about the error as possible, including the two character error code that is displayed.

4.1.2 User Errors

User errors occur when Access Manager finds avoidable problems; such as no more space on a disk or an illegal value for the KEY% parameter. You determine how Access Manager handles user errors by the value passed in the ERROPT% parameter via the INTUSR function. You have two options:

- If the INTUSR function is called with a nonzero value in the ERROPT% parameter, user errors can be trapped by testing the ERRCOD function for a nonzero value after calls to Access Manager functions. When a user error occurs, ERRCOD returns one of the values listed in Table 4-1.

- If the INTUSR function is called with a zero value in the ERROPT% parameter, the console displays a user error message of the following form:

  ```
  USER ERROR ...XY... CHECK ACCESS MANAGER MANUAL
  ```

  where XY is replaced by one of the codes listed in Table 4-1. Control then returns to the operating system.

In Table 4-1, the symbol * marks errors that send a complete error message to the console. The symbol ^ marks errors that are trapped by Access Manager only if your operating system supports extended BDOS error returns.

Table 4-1. Access Manager User Error Codes

Value	Code	Explanation
21	AE	Cannot write an index file record. Check to see if the disk or its directory is full.
23	AG	Cannot read index file record. You might be attempting to use a newly created index file was never closed properly.
24	AH	There is no more directory space on the disk. This occurs while trying to create a new index file.
25	AI	Access Manager could not find the name of the index file in the disk directory while attempting to close the file. This might occur if overlays destroy the Access Manager data areas.
30	AN	KEY% parameter value is out of range. The value must satisfy the following equation: 0 <= KEY% < NKEYS%
31	AO	The IDXNAME$ parameter value in an OPNIDX function contains unacceptable information. Check to see if the parameter contains a null value.
33	BA	You are attempting to reuse an index file number (KEY% parameter value) already assigned to an open index file.
34	BB	The KEYLEN% parameter value in an OPNIDX function exceeds the maximum allowable value of 48.
35	BC	The IDXVAL$ parameter has been initialized with an improper value. Check to make sure the parameter value is at least as long as the key length.
36	BD	While using the ADDKEY function, an attempt was made to assign zero as the data record number.

Table 4-1. (continued)

Value	Code	Explanation
39	BG	The key length (KEYLEN%) is too long for the number of disk sectors assigned (NNSEC%). The number of sectors must be sufficient to accommodate at least four key values.
40	BH	When trying to open an index file with the OPNIDX function, Access Manager found the file to be corrupted. Index files become corrupted when they are not closed (CLSIDX or SAVIDX functions) following updates. The index file must be rebuilt.
43	BK	Access Manager was unable to reopen an index file while performing the SAVIDX function. This error should not occur under normal circumstances. If it persists, notify Digital Research.
44	BL	You have attempted to close an index file that is not currently open.
46	BN	You have made an attempt to use an index file that is not currently open.
51	CC	Access Manager cannot write the data record into the data file. The disk or directory might be full.
52	CD	While using the READAT, WRTDAT, or RETREC function, you have attempted to use a data record number of zero, or a data record beyond the logical end of the data file. If applicable, check the way you have used the NEWREC function.
53	CE	Access Manager cannot read the data record your program has requested. You might be attempting to read a newly created data file that was not properly closed, read past the physical end of the data file, or read unwritten data.
54	CF	There is no more directory space on the disk where you are attempting to create a data file.
55	CG	Access Manager cannot close the data file specified in a CLSDAT function parameter. This might occur if an overlay destroys the Access Manager data areas.

Table 4-1. (continued)

Value	Code	Explanation
60	CL	The data file number specified in a FILE% parameter is out of range. The file number must satisfy this equation: 0 <= FILE% < NDATF%
61	CM	The FILNAME$ parameter value in an OPNDAT or OPRDAT function contains unacceptable information. Most likely cause is a null filename.
63	CO	You have attempted to use a data file number (FILE% parameter) already assigned to another data file.
69	DE	The NEWREC function attempted to reuse a data record in which the first byte did not contain FFH. (The RETREC function sets this byte.) You might be attempting to use a data file that has not been properly closed.
70	DF	The data file you have attempted to open with the OPNDAT function is corrupted. The file can be corrupted if updates are made but not posted with the CLSDAT or SAVDAT functions. The data file can be opened using the OPRDAT function.
73	DI	Access Manager cannot reopen a data file during a SAVDAT function. See discussion of user error 43/BK.
74	DJ	You have attempted to close a data file (FILE% parameter) that is not currently open.
76	DL	You have attempted to use a data file (FILE% parameter) that is not currently open.
83^	EC	A physical error occurred on a disk during input/output to an index file.
84^	ED	You have attempted to update an index file on a disk with Read-Only status.
85^	EE	You have attempted to update an index file with Read-Only status.
86^	EF	An index filename references a nonexistent drive, or the File Control Block is no longer active.

Table 4-1. (continued)

Value	Code	Explanation
87	EG	Index file opened by another process in a multitasking environment.
88	EH	FCB checksum error on index file close.
89	EI	Incorrect or missing password for an index file.
90^	EJ	OPNIDX attempts to create a second copy of file instead of simply opening existing file. Should not occur. Report to Digital Research.
91^	EK	An illegal character (?) in index filename.
92	EL	Operating system open file limit exceeded during index file open.
93	EM	Space in system lock list exhausted during index file operation.
113^	GA	Physical error on disk during data file I/O.
114^	GB	Attempt to update data file on Read-Only disk.
115^	GC	Attempt to update Read-Only data file.
116^	GD	Data filename references nonexistent drive, or File Control Block is no longer active.
117	GE	Data file opened by another process in a multitasking environment.
118	GF	FCB checksum error on data file close.
119	GG	Incorrect or missing password for a data file.
120^	GH	OPNDAT attempts to create a second copy of file instead of simply opening existing file. Should not occur. Report to Digital Research.
121^	GI	An illegal character (?) in data filename.
122	GJ	Operating system open file limit exceeded during data file open.
123	GK	Space in system lock list exhausted during data file operation.
145	IA	You have attempted to erase an index file (KEY% parameter) that is not currently open.

Table 4-1. (continued)

Value	Code	Explanation
146	IB	The filename of the index file you are attempting to erase cannot be found in the disk directory.
147	IC	The type of lock you have requested in a DLOCK% parameter is unacceptable.
153	II	You have passed an unacceptable value in a function parameter. This code usually indicates a problem with the interface between the application program and Access Manager.
154	IJ	Usually indicates a bad language interface and/or failure to call the INTUSR function in your program.
155	IK	The number of B-Tree nodes in an index file exceeds 65,535.
161	JA*	The value passed in the PROGID% parameter is out of range.
162	JB*	Access Manager cannot open the queue for a shared, multiuser module. Make sure a Resident System Process has been included as part of the GENSYS procedure.
163	JC*	Access Manager cannot open the queue for a user because the PROGID% parameter value is out of range.
175	JO	You have attempted to erase a data file (FILE% parameter) which is not currently open.
176	K@	Access Manager cannot find the name of the data file you are attempting to erase in the disk directory.
177	KA	You have passed an illegal value in the DLOCK% parameter.
183	KG	Same as error 153/II above.
184	KH	Same as error 154/IJ above.
209	MA	While using the SETUP function, you have passed a bad parameter value, or the buffer area is too small.

End of Section 4

Section 5
RECREATE Utility Program

Access Manager provides a utility program for recreating index and data files when necessary. Normally, the only time you need to recreate files is when they are vulnerable and the hardware loses power or the application program terminates abnormally.

The source code for the RECREATE program is provided in each language supported by Access Manager. Your Programmer's Guide contains instructions and examples for using the RECREATE program with each of these languages.

To simplify the recreation process, the RECREATE program uses a parameter file to describe the characteristics of the index and data files.

5.1 Recreate Parameter File

The parameters you place in this file tell the RECREATE program how to rebuild a particular index or data file. When you run the RECREATE program, it asks you to enter the name of the Recreate Parameter File to use. The filename is the only thing you are asked to enter.

You can modify the RECREATE program to automatically read the appropriate parameter file without any user intervention, or to be started automatically by the application program when error trapping detects corrupted files.

The Recreate Parameter File is the only place Access Manager explicitly represents the relationships between index and data files. There are four types of records in the Recreate Parameter File:

- Recreate Header Record
- Data File Records
- Index File Records
- Key Part Records

Table 5-1 illustrates the contents of each Recreate Parameter File record type.

Table 5-1. Recreate Parameter File Record Contents

Parameter	Meaning
	Header Record
NO.DATA.FILES%	The number of data files to be recreated.
NNSEC%	The size of index file records in sectors.
	Data File Record i
FILNAME$	The data filename.
RECLEN%	The data record length.
NO.INDEX.FILES%	The number of associated index files.
BEG.REC%	The number of first record to be scanned.
	Index File Record i-j
IDXNAME$	The index filename.
KEYLEN%	The length of index file key values.
KEYTYP%	The key type (alphanumeric or integer).
DUPKEY%	The duplicate key flag.
NO.KEY.PARTS%	The number of data record fields concatenated to form a key value.
BLANK.KEY.TO.NULL$	How to deal with blank key values.
	Key Part Record i-j-k
BEG.BYTE%	The starting byte of key part.
KEY.PART.LENGTH%	The length of key part.

In Table 5-1, BEG.REC% in the Data File Record specifies the first data record used for actual data. If BEG.REC% is zero, the RECREATE program automatically computes its value as the first data record to begin on or after the 129th byte of the file.

In the Index File Record, NO.KEY.PARTS% specifies the number of data record fields to be concatenated to form a key value. This information is used directly in conjunction with the Key Part Record. BLANK.KEY.TO.NULL$ determines whether or not to convert a key value equal to all blanks to a null string. A Y indicates conversion is necessary; a N means no conversion is required. Convert a blank string to a null key value when a blank field means there is missing data and there is no key value to be entered. Null key values are not added to the index by the ADDKEY function, but a blank string is.

In the Key Part Record, BEG.BYTE% specifies the byte position of a key part and KEY.PART.LENGTH% specifies its length. For example, if BEG.BYTE% is two and KEY.PART.LENGTH% is five, the data in the second through sixth bytes of the data record forms a key

part. Multiple key parts can be combined to form a complete key value. Note that the first byte of the record is considered byte position number one.

When you run the RECREATE program, the order of the duplicate keys is determined by the order in which they are present in the data records. This might not be the same order as before the recreation, due to the reclamation of deleted records.

Figure 5-1 illustrates the required order for records in a Recreate Parameter File.

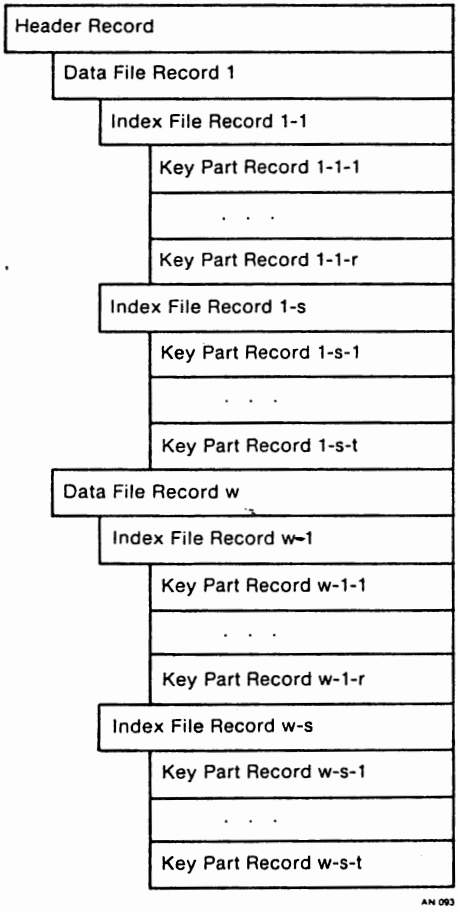

Figure 5-1. Recreate Parameter File Record Ordering

Figure 5-2 illustrates the contents of an example Recreate Parameter File. In this example, the header record indicates two data files and 512-byte index file records (NNSEC% equals four). CUSTOMER.DAT has a 100-byte record length and three associated index files. The RECREATE program automatically determines the first record with actual data because BEG.REC% is zero. Notice that the key parts for NAME.IDX and ZIPC.IDX are two bytes shorter than the key length declared in the respective index file records. This is because both of these index files are using the automatic suffixing of the key values to accommodate duplicates. If the key parts were two bytes longer, the suffix would replace the last two bytes anyway.

In Figure 5-2, INVENTRY.DAT has 192-byte records and only one associated index file. However, the key values for INVENTRY.IDX are constructed from three fields of the INVENTRY.DAT records: bytes 15-19, 100-101, and 4-6. That is, the ten bytes comprising these three fields are combined to form one key value.

```
2,4
CUSTOMER.DAT,100,3,0
NAME.IDX,10,0,1,1,Y
22,8
NUMB.IDX,4,0,0,1,N
2,4
ZIPC.IDX,11,0,1,1,Y
84,9
INVENTRY.DAT,192,1,0
INVENTRY.IDX,12,0,1,3,Y
15,5
100,2
4,3
```

Figure 5-2. Example Recreate Parameter File

5.2 Data File Recreation

The RECREATE program checks the integrity of each data file specified in the Recreate Parameter File. If a data file is not corrupt, the RECREATE program does not modify the data file. If the data file is corrupt, the RECREATE program corrects the data file in place. It reads each record in the data file to determine whether or not the record is active or deleted. If deleted, the RECREATE program automatically links the record onto the stack of deleted records for the data file. Once the entire file is read, the header record is rebuilt and written to the front of the data file.

A data record is considered deleted if the first byte of the record contains 0FFH. Access Manager automatically writes 0FFH into the first byte of each record deleted with the RETREC function. Therefore, if your application program uses RETREC to return deleted data records and either reserves the first byte of each record for the delete flag or ensures that no valid data can cause 0FFH in the first byte, you can use the RECREATE program as distributed.

If your application program conflicts with the use of 0FFH as a delete flag in the first byte of the data records, you must modify the RECREATE program to recognize deleted records some other way.

5.3 Index File Recreation

The Recreate Parameter File indicates which index files are related to each data file. If the data file corresponding to an index file is not modified by the RECREATE program, and the index file is neither corrupt nor empty, the index file is not recreated.

If the RECREATE program modifies the data file corresponding to an index file or the data file is corrupt, the index file is erased and recreated from scratch.

To speed recreation of the index file, the RECREATE program buffers scanning of the associated data file to extract key values. That is, the RECREATE program processes a large number of data file records before making the corresponding index file entries. This reduces disk head movement significantly. Further, the RECREATE program sorts the buffered key values before adding them to the index file. This also enhances the speed of the recreate process.

5.4 Recreate Messages

The RECREATE program generates two types of messages: those that report on the progress of the recreate process and those that report on unexpected conditions. Conditions that terminate the recreate process are caused by illegal values in the parameter file, lack of disk or directory space, inability to secure exclusive data file locks in a multiuser environment, or system inconsistencies.

If a system inconsistency arises (such as an inability to close a file or to reopen a file previously closed), be sure to check the integrity of your hardware and operating system. If these are operating correctly and there is no apparent cause for the RECREATE error message, you should go back to your most recent data base back-up files instead of trying to rebuild from your current, but corrupted, data.

End of Section 5

Appendix A
Access Manager Function Index

Table A-1 is an alphabetical list of Access Manager functions, cross-referenced to the mnemonic function name. These functions are organized alphabetically by mnemonic function name and discussed in detail in Section 3.

Table A-1. Access Manager Functions

Function Description	Name
ADD key to index file	ADDKEY
CHANGE data record number in index file entry	UPDPTR
CLOSE data file	CLSDAT
CLOSE index file	CLSIDX
COUNT entries in an index file	NOKEYS
COUNT nodes (used and available) in an index file	NMNODS
COUNT records in a data file	GETDFS
COUNT records used in a data file	GETDFU
DELETE key from index file	DELKEY
DETERMINE results of data lock/unlock request	LOKCOD
ERASE data file	ERADAT
ERASE index file	ERAIDX
FIND error code value	ERRCOD
FIND first entry in index file	FRSKEY
FIND two high-order bytes of data record number	DATVAL
FIND index entry equal to or greater than a specific key value	SERKEY
FIND index entry following a specific key value	AFTKEY

Table A-1. (continued)

Function Description	Name
FIND index entry in index file	GETKEY
FIND index entry preceding a specific key value	BEFKEY
FIND last entry in index file	LASKEY
FIND next available data record number	NEWREC
FIND next entry in an index file	NXTKEY
FIND previous entry in an index file	PRVKEY
INITIALIZE Access Manager	INTUSR
INITIALIZE Access Manager for single-user environment	SETUP
LOCK data file or data record	SETLOK
OPEN corrupted data file	OPRDAT
OPEN corrupted index file	OPRIDX
OPEN data file	OPNDAT
OPEN index file	OPNIDX
READ data record from data file	READAT
RECLAIM available data file records	RETREC
RELEASE data file or data record lock	FRELOK
SAVE data file updates	SAVDAT
SAVE index file updates	SAVIDX
SET two high-order bytes of data record number	SETDAT
WRITE data record in data file	WRTDAT

End of Appendix A

Index