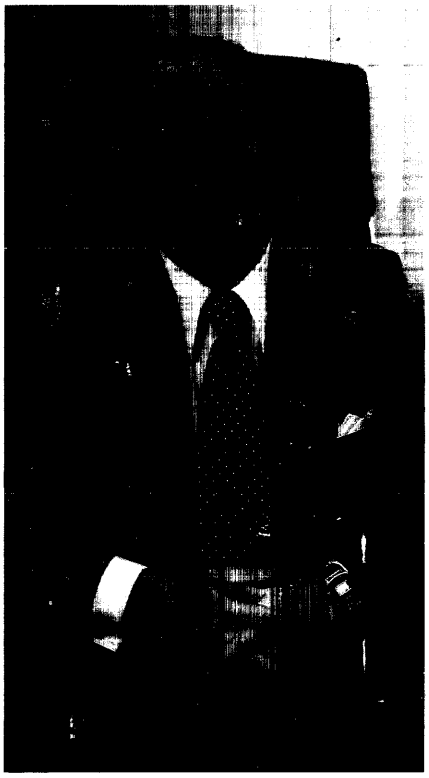
THE MIDAS USER'S GUIDE



PRIME
Computer

IDR4558

The MIDAS User's Guide
IDR 4558

This guide documents the software operation of the Prime Computer and its supporting systems and utilities as implemented at Master Disk Revision Level 17.6 (Rev. 17.6).

PRIME Computer, Inc.
500 Old Connecticut Path
Framingham, Massachusetts 01701

ACKNOWLEDGEMENTS

We wish to thank the members of the documentation team and also the non-team members, both customer and Prime, who contributed to and reviewed this book.

First Printing October 1980

All correspondence on suggested changes to this document should be directed to:

Laura Douros
Technical Publications Department
Prime Computer, Inc.
500 Old Connecticut Path
Framingham, Massachusetts 01701

CONTENTS

PART I. INTRODUCTION

PART II. CREATING A MIDAS FILE

PART III. MIDAS FILE ACCESS

ABOUT THIS BOOK...

This book is intended to replace PDR3Ø61, The MIDAS Reference Guide. Its emphasis is on acquainting the first-time or relatively new-to-MIDAS user with the basic concepts of MIDAS. However, the needs of the experienced or long-time MIDAS user have not been overlooked. Since it was not possible to satisfy everyone's particular needs at this first writing, anyone with useful ideas and suggestions regarding the improvement of the book should not hesitate to contact the author. All input will be considered for incorporation into future revisions of this book. This is your user's guide and only you know what else you'd like to see in it.

HOW TO USE THIS BOOK

Part I introduces MIDAS and briefly describes its major functions and features. Users needing additional background on terminology used here should see Appendix B.

Part II covers MIDAS file creation and the role of CREATK in setting up a MIDAS file template. MIDAS file deletion is also covered here.

Part III covers all the currently available language interfaces to MIDAS. It describes how to access a MIDAS file in COBOL, FORTRAN, BASIC/VM and so forth.

Part IV talks about the basic MIDAS file maintenance tasks that can be done by the user. It also covers MIDAS system maintenance and initialization, which is usually handled by the System Administrator, or whoever is in charge of MIDAS at a particular site.

Part V deals with the more esoteric aspects of MIDAS, most of which will mean little to new users at first. Veteran MIDAS users may find this material helpful in tuning MIDAS performance. Programmers who like to do things for themselves will find the information about the offline MIDAS routines useful in writing certain applications.

The collection of appendices includes a complete set of MIDAS run-time error message codes and explanations, a description of how MIDAS fits in with the rest of the PRIME file system, a summary of obsolete routines, and a summary of the changes that have occurred to MIDAS at this revision (17.6).

Conventions

The following conventions are used throughout this book, and should be reviewed before reading further:

Braces { }	Used to show a series of arguments, parameters or keywords, at least one of which must be chosen, unless the braces are themselves enclosed by brackets.
Brackets [ ]	Used to indicate that the enclosed parameters, arguments or keywords are optional.
Underlining	Generally used in examples to distinguish user input from system output. May also be used to indicate the minimum abbreviation of certain options, parameters, command words or responses.
(CR)	This symbol indicates a single carriage return, that is, a single hitting of the RETURN key on most terminals. It is used in examples to show that the user typed a carriage return, and nothing else in response to some MIDAS utility prompt.

Note

It is assumed that users already know that any command or response entered, either to PRIMOS or to any MIDAS utility, must be terminated by a carriage return, which causes the information typed by the user to be transmitted.

Part I
Introduction

SECTION 1

INTRODUCTION TO MIDAS

INTRODUCTION

MIDAS, the Multiple Index Data Access System, is a collection of subroutines and interactive utilities that enable the simple construction, access and maintenance of keyed data files. Because of their structure, information can be rapidly retrieved from MIDAS files through high-level languages like FORTRAN, COBOL and BASIC/VM. Once the overall shape (structure) of the MIDAS file is established, data can be added to it on-line through interactive programs, or through application programs. Alternatively, by using MIDAS utilities, data in existing sequential (non-MIDAS) files can be added to MIDAS files. The process of building a MIDAS file is quite simple. MIDAS does most of the work; the user merely provides the information requested by the various MIDAS file-building programs.

When To Use MIDAS

MIDAS file structure and access is key-oriented, making it easier for users to maintain, update and retrieve information stored in both large and small files. Although MIDAS is indeed a useful tool in many applications, it is not always the best answer. While this book cannot address every possible situation where MIDAS might be under consideration, it can offer a few basic guidelines. Briefly, MIDAS should be used when:

- A large file of information is to be accessed by one or more keys. For example, a customer master file may be accessed by account number or by customer name.
- A file needs to be accessed and updated on-line by several users simultaneously.
- Various factors like numbers of files per application, number of keys per file, number of files accessed per program, security requirements and necessary recovery features indicate that DBMS is not a preferred solution. These factors should be discussed with a systems analyst when determining which of Prime's data management products are most suited to your particular application.

How To Access MIDAS

MIDAS files are created and maintained by a collection of Prime supplied utilities and subroutines, all of which are later described in detail. MIDAS files can be accessed through these Prime languages:

BASIC/VM
COBOL
FORTRAN/FORTRAN 77
RPGII
PL/I
PMA

Ordinarily, a MIDAS file is set up for use with a particular language interface; however, it is possible to access a MIDAS file with any of these Prime high-level languages.

WHAT A MIDAS FILE LOOKS LIKE

Although the user need not know all the details of MIDAS file structure, there are some basic terms and concepts which are important. The remainder of this book assumes that the reader is familiar with certain fundamental information, most of which is dealt with in this section. Other details are covered in Appendix B, as indicated.

Some Basic Terms and Concepts

A typical data file is composed of records, which are divided into one or more related fields. Each field in a record is a piece of data, like a last name, age or phone number, which describes or pertains to an individual event, person, company, and so forth. Each record in the file has the same field layout; for example, each record has a last name, age, and phone number field. The important thing is that the actual contents of each field (called the field value) will generally differ from record to record. Thus, at least one field in each record will have a unique value, thereby distinguishing each record from all other file records.

Some file records contain two kinds of fields: those which identify the record and those which describe the record. The fields which identify a record are usually called key-fields or keys. Such fields are distinguished from other fields in the record which simply contain descriptive data, or "detail" information. Files with key fields are called "keyed" files. Detail information tends to be less unique or less used in everyday applications.

Keys are most often used for file searches because they readily identify a particular record. However, keys usually mean additional file overhead, that is, they require more storage. Keyed fields therefore should be used sparingly and chosen carefully. They should be fields which users are most likely to search on.

An Illustration: A keyed file, for example, might be an EMPLOYEE file which may contain many records, one for each employee in a company. Each data record contains information on each employee, and may be divided into several fields, such as SOCIAL-SECURITY-NO, LAST-NAME, FIRST-NAME, PAY-RATE, and EMPLOYEE-NO. (See Figure 1-1.)

Certain fields, like SOCIAL-SECURITY-NO, LAST-NAME, and PAY-RATE, are designated as file search keys.

A particular record is retrieved by searching on one of these unique key values. For example, to fetch the employee OGLETHORPE's record in the EMPLOYEE file (refer to Figure 1-1), you could use the LAST-NAME key value of "OGLETHORPE", or the SOCIAL-SECURITY-NO key value of "313-Ø9-8666".

Types of Keys: There are two kinds of keys: primary and secondary. A primary key must have a unique value for each record in the data file, with only one primary key field per record. A secondary key is not required to have a unique value in every record, and there may be as many as 17 secondary keys per MIDAS file record. Keys may be one of eleven data types. For complete details, see Section 2.

In the EMPLOYEE file, the SOCIAL-SECURITY-NO key could be the primary key because it guarantees a unique value for every employee in the file. The other keyed fields in the record, like PAY-RATE and LAST-NAME, are secondary keys and may or may not have unique values.

MIDAS File Structure

The concepts just described in the sample data file, EMPLOYEE, apply to MIDAS files as well. A MIDAS file is a type of keyed file known as a keyed-index file. A keyed-index file consists of at least two parts: a keyed data file and an index file.

The index file, called an index subfile in MIDAS, simply stores all the key values that appear in the MIDAS data file. Because there may be more than one key field in a data record, MIDAS creates one index subfile for each key. Because of this profusion of subfiles, a MIDAS file is also called a segment directory. A segment directory is physically made up of one or more segment subfiles, but logically, the segment directory is one file with a single file name. In a MIDAS file, one of these segment subfiles is the data subfile, the other, the primary index subfile. See Figure 1-2 for a representation of MIDAS file structure. For a more complete description of MIDAS structure, as well as other Prime file structure concepts, refer to Appendix B.

KEY	PRIMARY KEY	SECONDARY KEY 1	SECONDARY KEY n	DATA •••	••• DATA
KEY-FIELD	KEY-FIELD-VALUE	KEY-FIELD VALUE	KEY-FIELD-VALUE	FIELD-VALUE	FIELD-VALUE

TYPICAL DATA RECORD

FIELD	PRIMARY KEY	SECONDARY KEY 1	SECONDARY KEY 2	DATA-FIELD
FIELD-NAME	SOCIAL-SECURITY NO.	LAST—NAME	PAY-RATE	FIRST-NAME
RECORD 1	313-09-8666	OGLETHORPE	301.25	GREGORY
RECORD 2	132-34-6789	WONG	348.00	JANE
RECORD 3	002-49-6222	CHERRY	380.00	DAVE
RECORD 4	134-01-9999	LARSON	348.00	SUSAN
• • • • • •				
RECORD n-1				
RECORD n				

Figure 1-1. Employee Data File With Three Keys

Index Subfile Contents: Index subfiles store key field values that appear in the records of a data file and are used by MIDAS when looking for a particular key value during a file search. Because there are two types of keys, there are two types of index subfiles: a primary index subfile and a secondary index subfile. Every MIDAS file has a primary index subfile, and, depending on the number of secondary keys, an equivalent number of secondary index subfiles.

Assuming that every record in the data file has at least one key field, (containing a primary key value), the primary index subfile for that MIDAS file would list all of these values; in this case, the primary index subfile would have one entry per data file record. Depending on how many data records have a field entry for the corresponding secondary key, and on the interface used to add the data records, the number of secondary index entries will vary. This is described in more detail in the language interface sections.

Index Subfile Entries: Each entry in an index subfile consists of a key value from a particular record in the data file plus a pointer to that record. A pointer is a three-word address that tells MIDAS where to find the associated record. The entries in an index subfile are stored in a B-tree, a special type of binary tree. The data file records are not stored in the same order as the index entries, but that doesn't matter because each index entry always points to the right data record. The index entries are always stored in an orderly fashion; the entry with the lowest value is the first entry in the index, followed by entries of increasingly greater value.

The "Template" Concept

Although the exact number of files and subfiles varies, there are always two logical parts to a MIDAS file: the various index subfiles, and the data file, which contains the information to be accessed. Collectively, these two parts are called a template.

The data file, also called a data subfile, consists of records which can be referenced through the primary index subfile by specifying a primary key value. Each entry in the data subfile is "pointed to" by its unique primary key entry in the primary index subfile.

What is a Template?: A file template is a structural definition of a MIDAS file; it consists of all the index subfiles needed for key value storage, plus a few other subfiles and pointers. It is essentially an initialized (unpopulated = empty) MIDAS file.

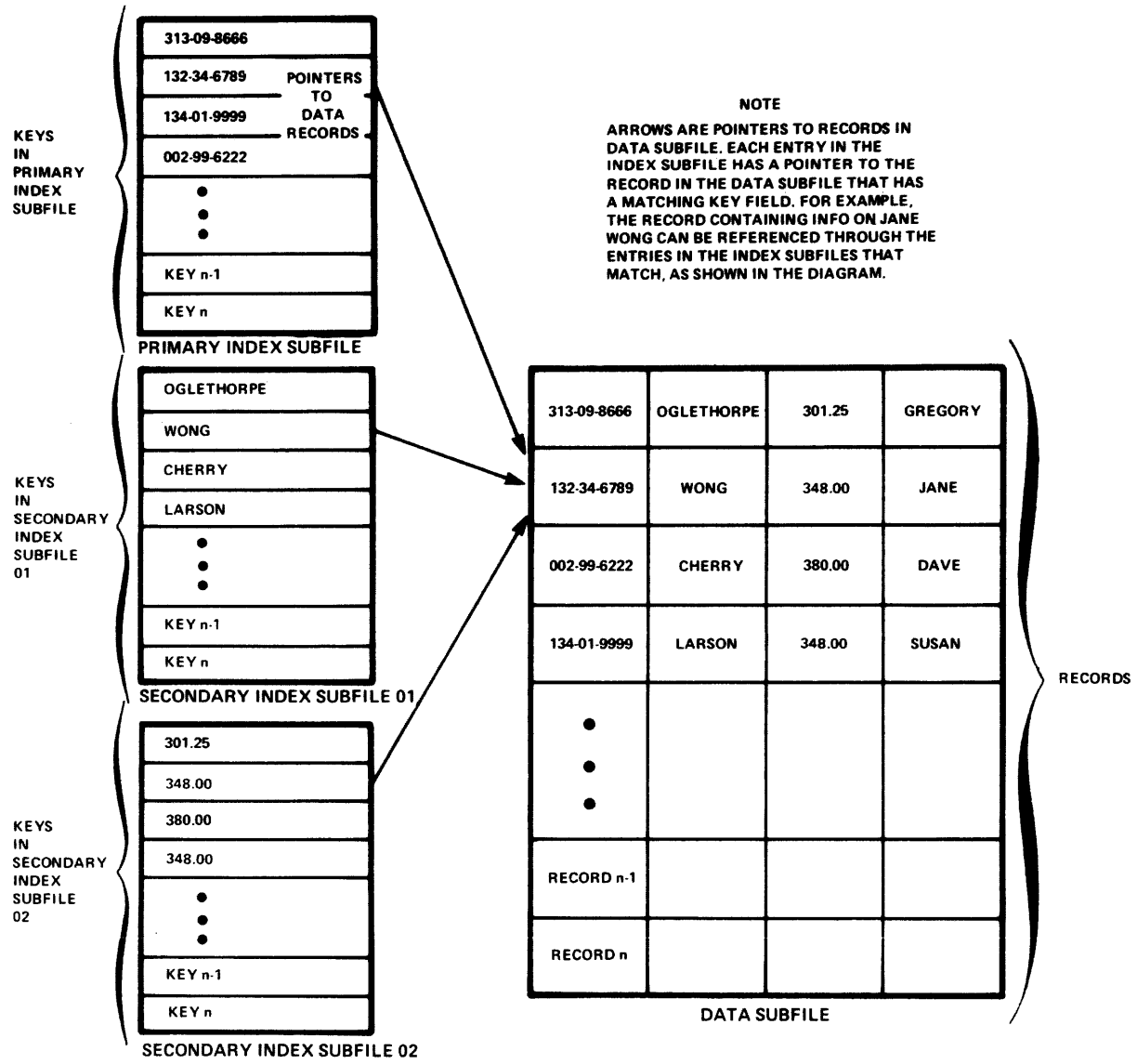


Figure 1-2. Logical View of a MIDAS File

A template's primary function is to accurately define the structure and properties of a MIDAS data file. MIDAS utilities and access routines require a template in order to access the information in a data file. The "definition" of a MIDAS file includes a description of all the data file's key types and lengths, and the data file record length.

Setting Up A Template: The MIDAS utility CREATK sets up a MIDAS file template using interactive dialog. CREATK asks for information about the template you intend to build. Basically, CREATK requires only the key types and sizes, plus the length of the data record, to be specified. Based on this information, CREATK sets up the appropriate index subfile skeletons and pointers required by MIDAS. (More on CREATK in Section 2.)

FROM THE USER'S VIEWPOINT

From the user's point of view, the first step in setting up a MIDAS file is to create a template with CREATK. The second step is to populate, or add data to the file, building the data subfile mentioned earlier. The data subfile can be "built" in several ways, all of which are discussed in Section 3. Once the file is populated you can access individual data records by primary and/or secondary key. Below are some descriptions of what happens during record addition and record access.

What Happens When You Add A Record

Records are added to a MIDAS data file by primary key; the primary key entry for the record is added to the primary index at this time. When you add a record to a MIDAS file in any of the language interfaces, the new record is written to the end of the data file. If desired, the user may then add the appropriate secondary key value to each of the secondary index subfiles for that file so the record can be referenced by one or more secondary keys. Keep in mind that there are no pointers from the data subfile to the index subfiles; pointers are maintained in one direction only: from the index subfile entries to the data subfile records. Only the pointer from the primary index subfile to the data subfile record is created upon record addition; secondary index subfile pointers must be added individually in some of the language interfaces. In others, the secondary index subfile entries are automatically added when the record (and primary index entry) is added.

What Happens When You Delete A Record

A record is "deleted" from a MIDAS file by deleting its primary key. Once a primary index entry (key value) is deleted, there is no way to get to the data subfile record it used to reference. The record in the data subfile is then marked as deleted, and the area is not reused until MPACK is run. MPACK is a special MIDAS utility that reclaims

"deleted" areas in the data subfile and in the various index subfiles for future use.

When a record is deleted by primary key, the corresponding secondary index entries for this record are marked for deletion but are not deleted until a user attempts to use them to reference this record. Since the pointer in the index entry references a deleted record, the secondary index entry is then removed from the index. However, secondary index entries can be deleted independently of the primary index entry in most language interfaces in which case the index entry space is automatically reclaimed (that is, MIDAS could use it for another entry).

Deleting Index Subfile: The entire contents and template description of a secondary index subfile can be deleted with the KIDDEL utility. KIDDEL asks for the numbers of the index subfiles to be deleted. The entire file can be deleted by specifying ALL. However, you cannot delete just the primary index subfile, as this would render the file inaccessible. See Section 4 for further details on the KIDDEL utility.

MIDAS File Access Methods

There are two ways to retrieve information from a MIDAS file: the "keyed-index" method and the "direct access" method. A MIDAS file template can be set up for keyed-index access only, or it can be set up to use both methods.

Keyed-Index Access: From the user's viewpoint, keyed-index access involves giving MIDAS a primary or secondary key value and waiting for MIDAS to return the appropriate record. MIDAS does keyed-index file searches by looking through a list of index subfile entries for a match on the user-supplied key value. Once a match is found, the corresponding record in the data subfile is located by following the pointer from the index subfile to the data subfile. Sequential searches are also possible by performing a "get next record" operation, which tells MIDAS to return the next record entry in the data subfile. Partial searches can also be done by using a prefix of the full key value.

Direct Access: Direct access is based on record numbers. Each record in the data subfile is given a unique number. To access a particular record, the user simply gives MIDAS a record number. Although the user must keep track of record numbers, this method can be faster than keyed-index access because there is less searching involved. Direct access files in COBOL require that the primary key be the record number. FORTRAN, however, does not levy this restriction; thus, direct access files in FTN can be accessed either by record number or by primary or secondary key. For direct access files with primary and secondary keys in addition to record numbers, the keyed-index access method can be used to retrieve information by key value. This means that the keyed-index method can be used on files of either type of template, while direct access only works on templates set up for direct

access. Direct access is supported only by the COBOL, FORTRAN and RPGII MIDAS interfaces.

THE MIDAS SYSTEM

The MIDAS system consists of Prime-supplied interactive programs, called "utilities," and file access subroutines. The utilities are responsible for file creation, modification and maintenance; the subroutines are used to add, delete, modify and access information in existing MIDAS files.

MIDAS Utilities

The MIDAS utilities are directly accessible from PRIMOS command level. These utilities build and modify MIDAS file structure:

- CREATK creates/modifies a MIDAS file.
- KBUILD adds entries to a MIDAS data subfile and index subfiles from sequential disk files.
- KIDDEL deletes a MIDAS file (partially or totally).
- MPACK restructures an overgrown or inefficient MIDAS file.

A Word on Utility Names: For those curious unbelievers, the MIDAS utility names do have some real meaning. CREATK is almost obvious -- the "K" stands for "keyed" to distinguish it from the other PRIMOS level command CREATE, which creates ordinary files and directories, but not segment directories, which is CREATK's job. KBUILD builds "keyed" files; KIDDEL deletes (DEL) keys, indexes and data (K, I, D); MPACK "packs up," or recovers space in MIDAS files (M). So, while they may not seem totally consistent, the names do make sense.

The MIDAS Life Cycle

For those of you who find it easier to think of dry subjects like MIDAS in biological terms, the following paragraphs are presented for your enjoyment.

The four MIDAS utilities described briefly above are in charge of managing the "life cycle" of a MIDAS file. CREATK gives birth to a MIDAS file, defining its shape and general characteristics. Based on the user's specifications, it creates a skeleton for the file. CREATK also serves as a "monitor" for the MIDAS file during its lifetime, providing important information about its skeletal structure, and, to some degree, about its contents. In fact, CREATK can even make minor adjustments to the file's skeleton at the user's command, improving its useability.

All this is well and good but we have to get information into the MIDAS file once we've defined its shape. The KBUILD utility does this by taking existing sequential (non-MIDAS) disk files with the information we want to put into the MIDAS database, and putting it into the proper cubbyholes in the MIDAS file superstructure. It puts key entries into the proper index subfiles and record entries into the data subfile.

Sometimes a MIDAS file begins suffering from the common ailments associated with repeated use (and abuse) and must undergo corrective surgery. The utility that performs this service is MPACK. It can do anything from a major overhaul to a simple face-lift. Major overhauls usually involve recovering the space wasted by data subfile and secondary index entries marked for deletion, plus checking to see that all index entries are in the proper order and reporting them to the user if they aren't, and finally, reordering the data subfile entries so the file can be used more efficiently. Minor cosmetic surgery involves unlocking records which were locked by program failure or maybe by system failure during file processing, or it might involve packing up an index subfile or two, again recovering space occupied by "deleted" index entries.

At some point in the life cycle of a MIDAS file, surgery may no longer be a viable option. Clearly, a miracle is in order. However, there is no life-giving utility in MIDAS. Instead, the KIDDEL utility has the unique task of meting out several types of death sentences to a MIDAS file. If the user determines that the shape of the MIDAS file can be used again in whole or in part, KIDDEL performs selective surgery on the file, removing parts of it entirely, or removing all the data entries from various parts of the file; this allows the file to begin a new life, reusing parts or all of its old skeletal structure. If, however, the MIDAS file has outlived its usefulness, KIDDEL will destroy it entirely, freeing up the disk space it formerly occupied.

This digression was intended to give you some idea of how the utilities work together as a package, and to make you more comfortable with the whole idea of MIDAS, which isn't such a frightful beast after all.

MIDAS File Access Subroutines

The remainder of MIDAS consists of data access and maintenance subroutines. These subroutines, listed and explained briefly below, have been integrated into and are used indirectly by the following languages: BASIC/VM, COBOL, PL/I and RPGII.

Special MIDAS access statement in those languages lets you obtain information from a MIDAS file without having to know anything about these MIDAS subroutines. FORTRAN and PMA users can call these subroutines directly from programs written in these languages. These subroutines are:

- ADD1$ adds an entry to a MIDAS file.
- DELET$ deletes a data file or index subfile entry.
- FIND$ locates a data file entry.
- LOCK$ finds and locks entry for exclusive access.
- NEXT$ locates next sequential entry in file.
- UPDAT$ updates/rewrites a file entry.

The calling sequences for these subroutines are discussed in Section 6.

LANGUAGE-DEPENDENT LIMITATIONS

Although any of Prime's languages may be used to access a MIDAS file, those languages with built-in interfaces have some limitations which must be kept in mind when using MIDAS. This is especially true if a file is to be accessed by programs written in more than one language. Below are the restrictions on template creation as they pertain to each language interface. Other restrictions pertaining to file access and maintenance are addressed separately in each of the language interface sections.

BASIC/VM

MIDAS files built for access by BASIC/VM programs can have up to 18 keys: one primary and 17 secondaries. Although keys are not required to be part of the data record, it is recommended that both primary and secondary keys be included in the data record for convenience. BASIC/VM does not support the direct access feature of MIDAS. See Section 8 for BASIC/VM information.

COBOL

A keyed-index MIDAS file, called an INDEXED SEQUENTIAL file in COBOL, can have up to six keys: one primary and five secondaries. If fixed-length records are desired, the data subfile record length indicated by the user during template creation should include the lengths of all primary and secondary key fields. This is because all keys must be included in the data record. Furthermore, the primary key must be the first field in the data subfile record. COBOL supports the direct access feature of MIDAS, enabling COBOL users to use direct

direct access feature of MIDAS, enabling COBOL users to use direct access MIDAS files, called RELATIVE files in COBOL, using the standard COBOL RELATIVE file I/O statements. See Section 7 for information on INDEXED SEQUENTIAL files, and Section 11 for details on RELATIVE files.

FORTRAN and PMA

Because FORTRAN is the principal MIDAS interface, and is in fact the basis of all the other language interfaces, FORTRAN and PMA users can take advantage of the full range of MIDAS features. Up to 17 secondary keys (and index subfiles) can be created per file. Keys do not have to be part of the data record, but it is easier to keep tabs on file integrity if they are. This simply means defining each key as an actual field in the record, as in the examples seen earlier in this section. See also Section 6.

RPGII

The RPG interface to MIDAS does not support the use of secondary keys or secondary data; only the primary key is recognized. Thus, records can be retrieved and reported on by primary key only. The size of the primary key is limited to 32 characters. Records in a MIDAS file can be updated or added through RPG, but cannot be deleted. RPGII supports access to both keyed-index and direct access MIDAS files. See also Section 10.

PL/I

The PL/I Subset G MIDAS interface supports only ASCII primary keys, with a maximum length of 32 characters. PL/I does not support these MIDAS features: secondary keys, secondary data or direct access. It is not necessary to use CREATK to set up a MIDAS file template, as PL/I has its own tools for doing so. However, files created with CREATK can be accessed through PL/I. See Section 9 for details.

Part II
Creating a Midas File

SECTION 2

INITIALIZING A MIDAS FILE (CREATK)

INTRODUCTION

MIDAS file creation involves two essential steps:

1. Initializing -- defining a template for the file to allocate space for all index and data subfiles.

2. Building/converting a data file -- adding data to the file (see Section 3 (KBUILD), and also Sections 5-1Ø).

This section deals with MIDAS file initialization, describing how to use CREATK in setting up a simple MIDAS file. Other CREATK functions are treated primarily in Sections 12 and 15.

Language Interface Dependencies

If you're getting acquainted with MIDAS for the first time, the importance of the language interfaces, to which Section 1 made reference, may not be immediately apparent. The MIDAS utilities provide all the essential tools for building, maintaining and destroying a MIDAS data base, but they don't provide any tools for information extraction. That's why the language interfaces are essential. They enable you to access (read), update, delete and add new information to MIDAS files. Without them, a MIDAS file would be a mass of inaccessible information.

Each language interface to MIDAS works a bit differently; some support all the features of MIDAS, and others, only a few. Therefore, it's important to know what language interface you'll be using to access a file before you create that file; that way you'll be able to tailor the file to the requirements of that particular language interface. These requirements and guidelines are all summarized in Table 2-1.

SETTING UP A TEMPLATE

CREATK is an interactive program that sets up a template based on user-supplied file specifications, often called "parameters," which include:

- MIDAS file type (keyed-index or direct access)

- Primary key type and size

Table 2-1. Summary of Interface Requirements

Language Interface	Primary Key	Secondary Keys	User Data Size (record-length)
BASIC/VM	not required to be in data record, but is recommended for ease of use*	up to 17 (numbers 1-17)	enter (CR) or Ø for variable-length records; or enter record size in words for fixed-length records
COBOL	must reside in data record and must be first field in record	up to 5 (numbers 1-5) must be defined in order	must include lengths of all key fields (in addition to non-key fields) if fixed-length records are desired; variable-length records are supported, but keys must reside in record.
FORTRAN/ PMA	not required to reside in data record (but recommended)*	up to 17 (numbers 1-17)	enter size in words for fixed-length records; enter (CR) or Ø for variable-length records
PL/I (see Note)	must be an ASCII key; default size is 32 chars. -- does not have to be part of data record but is recommended for ease of use*	no secondary keys supported	Variable-length records only, if creating file from PL/I program; fixed-length records supported if file was created with CREATK
RPGII	must reside in data record; does not have to be first field in record	no secondary keys allowed	fixed-length records only; size must include primary key

Note

PL/I programmers do not have to use CREATK when initializing a MIDAS file template. PL/I can create one for you: see Section 9 for details.

* Files with keys resident in the data record can be easily rebuilt with KBUILD. See Section 3 for details.

On page 2-3, the first sentence should read:

Secondary key types and sizes -- these are optional and should be used when you want more than one search key for the file. Secondary search keys do not have to be part of the data record except in COBOL.

- Secondary key types and sizes -- these are optional and should be used when you want more than one field per record to use a search key; the maximum number of secondary search keys allowable depends on the language interface which will be used to access the file. (See Tables 2-1 and 2-3 for more information on keys and key types).

CREATK's Dialogs: Minimum Options vs. Full Options

CREATK has two dialogs:

- The "minimum options" dialog, which lets MIDAS supply default values for most of the structural parameters needed to build the template

- The "full options" dialog which lets the user provide these parameter values

With either CREATK dialog, the user can set up a keyed-index or direct access MIDAS file, depending on the access method desired.

The real difference between the two dialogs is that the "full" (also called "extended") options path requires the user to supply more information. Under minimum options, a user can set up either a keyed-index or direct access file by answering a few simple questions about the file and its keys. The full options dialog, which asks the user for file size details like segment length and index block size, is discussed in Section 15.

Generally, the minimum options path provides all the options most users need. Full or extended options features are required only when the user wishes to increase or decrease the index block size.

Minimum Options: Keyed-Index Dialog

The remainder of this section explains the use of CREATK's minimum options path to create a keyed-index MIDAS file. If you want information on creating a direct access file, see Section 11. The extended options path is covered in Section 14. The complete dialog for creating a keyed-index file appears just below this discussion.

All input to CREATK can be in lowercase or uppercase and must conform to the guidelines explained in Table 2-2. In cases of improper input, CREATK usually displays an explanatory message of some sort, telling you what it's looking for. This is repeated until a proper response is given.

Table 2-2. Minimum Options (Keyed-Index) Dialog

Prompt	Response
MINIMUM OPTIONS?	YES (for simplest options path)
FILE NAME?	Enter pathname of file to be created.
NEW FILE?	Enter YES to create new template.
DIRECT ACCESS?	Enter NO to create keyed index file.
DATA SUBFILE QUESTIONS	
PRIMARY KEY TYPE:	Define primary key type; enter one of the key codes listed in Table 2-3, below.
PRIMARY KEY SIZE=:	Size can be specified with B nn, where nn is number of bytes for an ASCII key or the number of bits for a bit string key; size can also be given in number of words, W nn, for ASCII or bit string keys. See Table 2-3.
DATA SIZE=:	For fixed-length records, indicate maximum length of data record in the data subfile; size is expressed in words. The length of all keys must be included in the data size for MIDAS files to be used in COBOL applications. Enter (CR) or Ø for variable-length records.
SECONDARY INDEX	
INDEX NO?:	Enter number from 1-17 (FTN and BASIC/VM) or 1-5 (COBOL), indicating which secondary index is being defined. In COBOL, secondary keys must be defined in order, that is, secondary key 1 must be defined before secondary key 2, and so forth. Enter Ø or (CR) to terminate secondary index definition sequence.

DUPLICATE KEYS PERMITTED?	Answer YES or NO. YES allows the same key value to appear in more than one record. Duplicate values are legal for secondary keys only.
KEY TYPE:	Enter one of codes listed in Table 2-3.
KEY SIZE=:	Enter size of key in words, bytes or bits. See Table 2-3.
SECONDARY DATA SIZE=:	Enter number words of secondary data to be stored with this secondary key; for FORTRAN and PMA only. Otherwise, enter Ø or (CR).

Table 2-3. MIDAS File Key Types

KEY CODE	KEY TYPE	Length Specification
A	ASCII	Words or Bytes: W nn or B nn Max. 32 words (64 bytes)
B	Bit string	Bits or Words: B nn or W nn Max. of 16 words (32 bytes)
D	Double Precision Floating Point	Hardware-defined: 4 words
I	Short Integer (INT*2)	Hardware-defined: 1 word
L	Long Integer (INT*4)	Hardware-defined: 2 words
S	Single Precision Floating Point	Hardware-defined: 2 words

When CREATK asks a "YES/NO"-type question, it will accept any one of these possible responses:

YES
NO
AYE
NAY
OK

These responses can be abbreviated to one letter and can be typed in uppercase or lowercase.

The CREATK dialog terminates when the user hits (CR) in response to the INDEX NO? prompt.

Key Types: The data types for MIDAS keys are listed in Table 2-3. The maximum number of words per key is limited to 16 words for bit strings and 32 words for ASCII strings. The other data types are automatically sized according to their internal specifications, as shown in the table.

A Sample File

To help illustrate the basic features of CREATK, Figure 2-1 shows the layout of a sample MIDAS file which is used throughout the book to illustrate various MIDAS concepts. This sample file is called CUSTOMER, and will contain information about certain vendors, including names, locations and regions. It could also contain order information, a sales rep name, etc. For simplicity, the file record has been limited to three major fields: customer code, a unique field which identifies each customer, the customer name field and the region code. The record can be expanded to accommodate more data, like the customer's address.

The CUSTOMER file is a minimum options keyed-index file created with one primary key and two secondary keys. The primary key is an ASCII key, five characters in length; it describes the customer code, which is a unique field in each file record. The first secondary key is an ASCII key of 25 characters referencing the customer name field of each file record. The second secondary key, a four character ASCII string, describes the region-code field. It will be comprised of a two-letter code describing one of six geographic regions:

MW	mid-west
NE	north east
NW	north west
SE	south east
SW	south west
WR	western region (California, Nevada, etc.)

and a two letter state code, taken from the standard two letter abbreviations for state names.

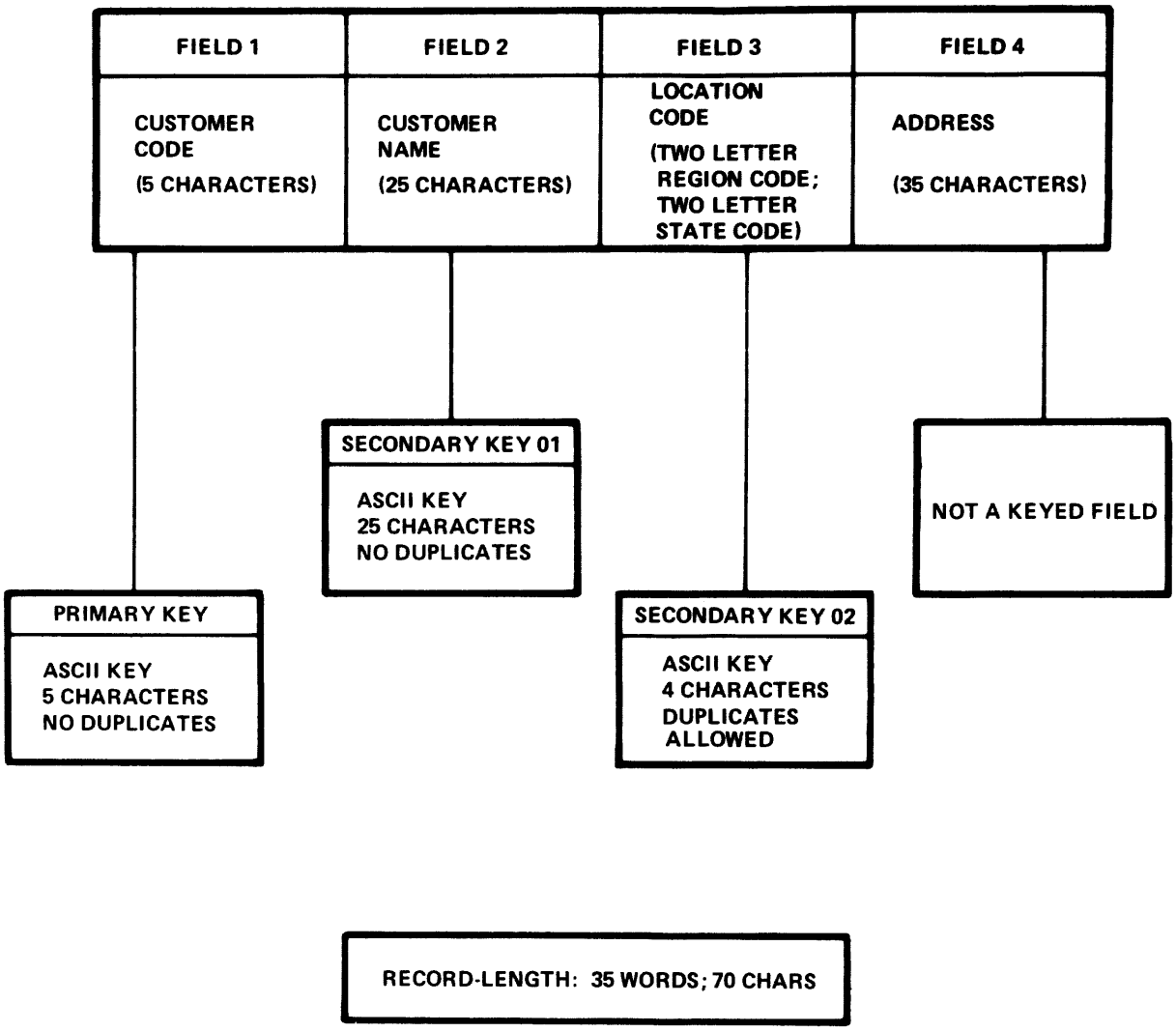


Figure 2-1. Layout of CUSTOMER File

Sample CREATK Dialog

To show you how the CUSTOMER file template could be set up with CREATK, a comoutput file (PRIMOS command output file), containing CREATK dialog prompts and responses is listed below. The responses entered at the terminal by the user have been underlined to distinguish them from CREATK's prompts. This is only a text convention and is used only for clarity. Don't attempt to underline your input to CREATK (or to anything else, for that matter). Note that your responses can be entered in uppercase or lowercase letters.

Because we want to access the file through all the language interfaces, we chose fixed-length records so there would be no ambiguities. Thus, the USER DATA SIZE is supplied as 35 words, giving the data file fixed-length records of 35 words (70 characters) each.

Important Observations: The secondary data feature is not used, therefore a carriage return was supplied for this prompt. The use of secondary data is not recommended, and in fact is highly discouraged; see Appendix C. The convention for representing a carriage return in this document is (CR). Understand that each line input to all MIDAS utilities must be terminated by a carriage return. The symbol appears only where it's necessary to point out that a "null" response was entered to a particular prompt. User input is underlined in all examples in this book to distinguish it from system output.

```
OK, creatk
[CREATK rev 17.6]

MINIMUM OPTIONS? yes

FILE NAME? customers
NEW FILE? yes
DIRECT ACCESS? no

DATA SUBFILE QUESTIONS

PRIMARY KEY TYPE: a
PRIMARY KEY SIZE = : b 5
DATA SIZE = : 35

SECONDARY INDEX

INDEX NO.? 1

DUPLICATE KEYS PERMITTED? no

KEY TYPE: a
KEY SIZE = : b 25
SECONDARY DATA SIZE = :  (CR)

INDEX NO.? 2
```

```
DUPLICATE KEYS PERMITTED? yes

KEY TYPE: a
KEY SIZE = : b 4
SECONDARY DATA SIZE = :  (CR)

INDEX NO.? (CR)

SETTING FILE LOCK TO N READERS AND N WRITERS

OK,
```

OTHER THINGS TO KNOW ABOUT CREATK

CREATK is a complex utility, capable of performing many different tasks. So far this section has only given you a sampling of its repertoire. The next few pages describe some important things about CREATK that will help you use MIDAS and that will increase your understanding of how CREATK works and what it does.

File Read/Write Locks

By default, CREATK sets the file read/write lock on each MIDAS file it creates to n readers and n writers. ~~This is equivalent to the PRIMOS RWLOCK setting of 3.~~ With a lock setting of 3, it is possible for one or more users to have the file open for reading, while one or more users have it open for writing or updating. These default read/write lock settings are part of MIDAS's concurrent process handling method, described in Section 13.

CREATK displays the message:

```
SETTING FILE LOCKS TO N READERS AND N WRITERS
```

at the end of every session in which a new MIDAS file is created.

Page 2-10 states that a read-write lock setting for n readers and n writers is "equivalent to the PRIMOS RWLOCK setting of 3". This is incorrect. Instead, what was meant was that you would use the FUTIL "3" setting to set the file read-write lock to n readers and n writers.

Note

If you use FUTIL to TRECOPY a MIDAS file from one location to another, the read/write lock settings of the file are reset to the system default. Remember to reset the locks on these files to 3 before attempting to access them from MIDAS. The lock on a particular file can be set with the SR subcommand. Simply type:

```
SR filename 3
```

where filename is the filename of the copied file.

More CREATK Functions

CREATK has many other options which allow you to obtain information about an existing MIDAS file, its key types and sizes, its index subfile structure, segment length, block size, etc. It is possible to add or modify new index subfiles. You can change the length of the data file record (expand it preferably) and get estimates on how much room is needed for a projected number of entries, given a certain file layout. To obtain a list of all these "old file" options, type H (for HELP) in response to the FUNCTION? prompt. These options can only be used on existing files, i.e., when you answer "no" to the "NEW FILE?" prompt:

```
OK, creatk
[CREATK rev 17.6]

MINIMUM OPTIONS? yes

FILE NAME? customers
NEW FILE? no

FUNCTION?
```

See Section 12, MIDAS FILE MAINTENANCE, for details on all the "old file" options of CREATK, which are summarized below for your convenience.

Summary of CREATK Options

Once you've indicated to CREATK that a file is an old or existing file (by entering "no" to the NEW FILE? prompt), CREATK asks which FUNCTION you want. The entire list of functions can be displayed by typing "help":

```
FUNCTION? help
  A[DD]       =  ADD AN INDEX
  D[ATA]      =  CHANGE DATA RECORD SIZE
  E[XTEND]    =  CHANGE SEGMENT  SEGMENT DIRECTORY LENGTH
  F[ILE]      =  OPEN A NEW FILE
  H[ELP]      =  PRINT THIS SUMMARY
  M[ODIFY]    =  MODIFY AN EXISTING SUBFILE
  P[RINT]     =  PRINT DESCRIPTOR INFORMATION
  Q[UIT]      =  EXIT CREATK
  (C/R)       =  IMPLIED QUIT
  S[IZE]      =  DETERMINE THE SIZE OF A FILE
  U[SAGE]     =  DISPLAY CURRENT INDEX USAGE
  V[ERSION]   =  MIDAS DEFAULTS FOR THIS FILE
```

The brief descriptions provided by CREATK are expanded a bit in the following paragraphs. For details on these functions and how to use them, see Section 12.

ADD: Allows you to add a secondary index subfile (and a key) to an existing MIDAS template. It does not let you add more than 17 secondary index subfiles and won't allow you to specify index subfile 17 as double-length (if you've enabled this feature: see Section 15).

DATA: Changes the data record length and the number of records allocated for that file if it is a direct access file; it does not display the current record length, so it must be known (use the PRINT option to get it: see below).

EXTEND: Lets you change the number of segments per segment directory and words per segment subfile. See Section 15 for more details. This effectively permits you to make bigger index and data subfiles by extending the segment subfile and segment directory lengths.

FILE: Essentially re-starts CREATK; lets you create a new file template without returning to PRIMOS and re-entering CREATK, or work on another old file. Returns you to PRIMOS after file definition is complete.

HELP: Displays the list of functions as shown above.

MODIFY: Allows you to change support of duplicates; changes secondary data size; changes single to double-length index, if double-length indexes are supported. Also allows you to change the index block length, if using the extended options version of CREATK. See Section 15.

PRINT: Describes each index subfile and the data subfile in terms of segments allocated, index capacity, key type, key size and number of index levels for that subfile; for each level, it describes the entry size, block size, control words, maximum number of entries per block and the number of blocks in that level. Data subfile information displayed, as of last MPACK, includes the file access type (keyed or direct), the number of indexes, the entry size and the key size. See also Section 12.

QUIT: Exits the CREATK dialog and returns to PRIMOS. A carriage return does the same thing.

SIZE: Estimates the number of segments and disk records required for a hypothetical number of entries; estimates can be made for each index subfile, for the data subfile and the total file. See also Section 12.

USAGE: Provides information on the total number of entries in the file, the number of entries indexed, and the number of entries deleted, and the number of entries inserted since last MPACK; also displays the version of MIDAS which last modified the file. See also Section 12.

VERSION: Displays the rev. stamp of the version of MIDAS under which this particular file was created; also displays the default parameters for the file (DAM file length, segment directory length, segments index, etc.). See also Section 12.

There are several new messages output by KBUILD during the process of building a file. Most of them should not concern the user as they are simply informative and do not indicate difficulties. The diagnostic messages make more sense if you understand how KBUILD goes about building a MIDAS file. Briefly, KBUILD builds a MIDAS file in one or more stages, called "passes." On each pass, one or more index subfiles are built or are deferred for building during a subsequent pass. The KBUILD message:

```
FIRST BUILD/DEFER PASS COMPLETE
```

simply indicates that KBUILD has finished building the data subfile, (if the primary index needs to be built) and has built one or more indexes while possibly deferring the building of others. KBUILD defers the building of an index only if the index to be built is empty and the user-provided input is unsorted. During the first pass KBUILD puts the unsorted input entries for each such index into a temporary "defer" file. After the first pass is complete, KBUILD sorts the individual defer files and builds the index subfiles from the now sorted data.

Each time an index is sorted, KBUILD prints out a message indicating which index it is going to sort. After the sort is complete, the message:

```
SORT COMPLETE
```

is printed. Similarly, KBUILD announces the building of each index. For example:

```
BUILDING INDEX 0
```

After this index is built, the message:

```
INDEX 0 BUILT
```

is displayed. When KBUILD has finished building the data subfile and all the index subfiles that needed to be built, it displays the message:

```
KBUILD COMPLETE.
```

and control returns to PRIMOS (unless you are running KBUILD out of a command file).

SECTION 3

BUILDING A MIDAS FILE (KBUILD)

INTRODUCTION

Throughout this book, the terms "building," "data entry" and "populating" all refer to the process of adding data to a MIDAS file. Once you've set up a MIDAS file template there are several ways to add data to, or populate it. The alternatives are summarized below, along with references to other sections in this book where you can find more information on populating MIDAS files.

There are at least three ways to build a file (four, if you're programming in BASIC/VM). One of these ways, the KBUILD utility, is documented in this section. The other methods:

- Application Programs
- Interactive Entry
- Offline Routines

are covered as indicated in Table 3-1. A quick comparison of these methods is also included to help you decide which method is best suited to your needs.

When to Use KBUILD

KBUILD is a quick and easy-to-use method of building a MIDAS file. With it, you can build both data and index subfiles from sequential disk files. KBUILD can also be used to add entries to a secondary index subfile, using entries in a sequential disk file or entries already present in the MIDAS data subfile associated with this index.

With KBUILD you can build keyed-index MIDAS files containing either fixed- or variable-length records, and you can build direct access MIDAS files (which always have fixed-length records). Special direct access information is summarized at the end of this section under KBUILD AND DIRECT ACCESS.

Any of these file types can be processed by KBUILD:

- ASCII text files (compressed)
- PRWF$$-written binary files
- FORTRAN-written binary files
- COBOL-written "text" files with primary key as first field (uncompressed)
- RPG-written files (uncompressed)

These file types are further described in Table 3-2.

When to Use a Program

Application programs to build MIDAS files require more work of the user than does KBUILD. They should be used only when one or more of the following is true:

- You don't have a pre-existing sequential disk file containing data in an easily convertible form.
- It would require more work to prepare an existing data file for KBUILD than it would to simply use ADD1$ (FORTRAN call interface), one of the "add" statements in the other language interfaces, or other offline routines.
- A particular application program has already been written to handle updates, deletions, etc., and it would not require much effort to convert or modify it to perform record addition.
- A series of finds, updates and adds must be done on a regular basis and it would be easier to take care of everything with a single applications program.
- Other users may need access to the file even while another program is adding entries to it. The only way to do this safely is to use the online routines like ADD1$, either directly (through FORTRRAN or PMA) or indirectly through COBOL, BASIC/VM, and so forth. KBUILD and the other offline routines (discussed in Section 14) must have single-user access to a file while building it. However, they cannot guarantee that this remains true during the entire build process, so if another user or process accesses the file while one of these routines has it open, the file will be damaged.

Data entry as implemented in the various language interfaces is covered in Sections 6 through 10.

Table 3-1. File-Building Alternatives

Alternative	When to Use	Where to find Info
KBUILD (interactive utility)	Mainly for COBOL, RPG and FTN programmers who already have existing data in sequential disk file. KBUILD adds entries from these input files to the index subfiles and data subfile of the MIDAS file template. Data in the input files must include a primary key and all records in the file must be formatted identically. Files can be in text, binary, COBOL or RPG-type format: details below.	Section 3 (this one)
Application Program	Use this method when existing disk files are not in the form required by KBUILD, or when only a small number of records are being added.	Sections 6-10
Interactive Entry	For BASIC/VM only; when you want to add a few entries without a lot of programming hassle.	Section 8
PRIBLD, SECBLD, BILD$R (offline routines)	Users can call offline routines PRIBLD, SECBLD, and BILD$R to build a MIDAS file from existing sequential disk files; useful for adding concatenated keys or secondary data. See Section 14 for details.	Section 14

Table 3-2. File Types Supported by KBUILD

File Type Code	Description
BINARY	A binary file created by PRWF$$, which is usually called from a FORTRAN program. There are no newline characters (.NL.) in such a file.
COBOL	An ASCII file created by O$AD08, a routine called by COBOL as a result of a WRITE statement. The primary key must be the first field in the record. These are uncompressed files with fixed-length records and newline characters as delimiters.
FTNBIN	A binary file created by a FORTRAN WRITE statement via the routine O$BD07, which is used in FORTRAN binary output. The first word of each record in this type of file indicates the record-length of that record. Contains no newline characters.
RPG	A file created by the O$AD08 routine; an ASCII uncompressed file with fixed-length records and newline character delimiters. Records must contain the primary key but it doesn't have to be the first field in the record.
TEXT	An ASCII file (written with O$AD07) that has been passed through the EDITOR and is thus compressed. The records may be variable-length and are terminated by a newline character.

USING KBUILD

The remainder of this section describes all the important features of KBUILD, familiarizes you with the basic KBUILD dialog and variations thereof, and shows you how to use KBUILD in some typical situations.

KBUILD's Functions

KBUILD's range of functions are summarized below:

- Adding data to a new ("empty") MIDAS file template and building (adding entries to) the necessary index subfiles from sorted or unsorted input data.
- Adding new data and index entries to a MIDAS file that already contains data entries.
- Adding entries from an external data file to a newly created secondary index subfile which has just been added to an existing (and previously-populated) MIDAS file.
- Converting a field from existing MIDAS data subfile records to a secondary key field and adding these entries to a new or already existing secondary index subfile.

Most frequently, KBUILD is used when a large quantity of records need to be added to a MIDAS file. For each record, KBUILD takes care of adding the primary index entry, the data subfile entry and any secondary index entries that have been supplied. Especially in cases where lots of entries are being added to a file, it is recommended that you keep a "backup" copy of the input file or files in case of damage to the MIDAS file, or in case of a system crash while the file is being processed. Set up a command file that first invokes CREATK to set up the template, then invokes KBUILD to populate the file.

The user must know the following to use KBUILD:

- The record structure of the input files (this shouldn't be a problem since the user will most likely set up the input file to suit the situation at hand)
- The record size definition of the MIDAS data subfile as supplied to CREATK

Note

KBUILD does not process secondary data (it zeroes it out) and cannot handle concatenated keys. Use PRIBLD and SECBLD to build a file with these features: see Section 14.

File Types Supported by KBUILD

KBUILD supports input files with fixed-length records only. However, MIDAS files with variable-length records can be built by KBUILD, using specially formatted input files, as explained further under Variable-Length Records, below. Each file type supported by KBUILD is identified by its own type-code, as shown in Table 3-2, above. It's important that you know the type of file you're using before entering the KBUILD dialog, because KBUILD cannot process the input file without this information.

Input File Rules

Input files always have fixed-length records, regardless of whether the output (MIDAS) file contains fixed- or variable-length records. The format of the input file record varies, depending on what you intend KBUILD to do with the file. Input files can contain:

- Primary key values and data values -- secondary key values optional
- Secondary key values only: must include primary key value in each input record so the record to be referenced can be located

"Data" means the information that is to be written to the data subfile. Records do not have to be in sorted order, although it is recommended that they be sorted by any or all keys for faster processing. Some users may have existing files that they effectively want to "convert" into MIDAS files by using KBUILD. These files may or may not contain extraneous information that the user doesn't want processed. In any event, it's important to note that KBUILD expects certain things of input files and their record structure. The data record structure should be roughly laid out as follows:

- Each record of the input file must begin with the pertinent data to be transferred by KBUILD to the appropriate parts of the MIDAS (output) file. In other words, the data to be placed in the index subfiles and in the data subfile must appear in the front of the input record.
- No extraneous data can appear before the fields you want KBUILD to process. This restriction makes sense because it is easier for KBUILD to truncate the part of the record it doesn't need than to extract bits and pieces from a jumbled-up record.

- KBUILD requires the user to specify the starting character position of each key field in the input record; this is also called the "byte offset" of the key field. The first character position in the record is character position 1, not Ø. Key fields must begin on byte (half-word) boundaries; they are not required to precede the data portion of the input record, except for the primary key in COBOL. They may appear after the data if they are not going to be physically part of the data subfile entry.

- KBUILD requires the user to tell it the input record length. This length is the number of words in an input record, not including anything inserted by the file system, like .NL. characters or leading word count (in FTNBIN files). The user doesn't have to tell KBUILD the length of the output file record, since it figures that out by reading the MIDAS file configuration.

Note

For COBOL type files only, the primary key MUST begin in record position 1 (byte offset 1) of each input file record. RPG files do not require that the primary key start in position one.

Where Keys Should Reside

Should you not want the keys to reside in the data subfile records, put them AFTER the data you want included in the data subfile entries. KBUILD can then truncate them when it writes the entries to the data subfile. Only the initial portion of the input record (without keys) will be written to the MIDAS data subfile. Remember, in COBOL the primary key must begin in record position 1 but secondary keys can appear anywhere in the record.

Record Compatibility

Each record of the input file should have exactly the same record layout; that is, if the primary key starts in character position 1 of the first record, it should begin in the same position in each of the subsequent records in the input file. The same goes for any secondary key fields. Furthermore, it is assumed that the portion of each input record to be written to the output file always begins with word 1 of the input file record. When writing a record from the input file to the output (MIDAS) file, KBUILD always begins with word 1 (character position 1) of the input record. The actual length of the entry written to the output file depends on whether the MIDAS file has fixed-length records or variable-length records. For more information on this topic, see Building a MIDAS File With Variable-Length Records, below.

Sample Record Layout

The following paragraphs and illustrations should give you an idea of record layouts for applications that require keys to reside in the data subfile record (like COBOL), and for applications that exclude keys from the data record.

In the record layout pictured in Figure 3-1a, the primary key starts in character position (byte offset) 1. Secondary key Ø1 starts in position 8. KBUILD knows how long the key is supposed to be by looking at the MIDAS file. Secondary key Ø2 starts in character position 15. The non-key portion of the input record begins at position 21 and would continue for as many words as the user specifies. This record layout represents the kind of input file structure you'd need to include the key fields in the data subfile record. Each record in this input file would follow the same pattern.

Keys Not Resident in Data: If keys were not included in the data subfile record, the record layout might look something like that shown in Figure 3-1b.

The data portion of the record is 2Ø characters long; the primary key begins in character position 21; secondary key Ø1 starts in position 28, and secondary key Ø2 starts in position 35. After the key values are written to their respective index subfiles, the first 2Ø characters (1Ø words) of the input record are copied to the data subfile.

What's Next

Most of the general things you need to know about KBUILD have just been covered; the next part of this section discusses the particulars of using KBUILD, like:

- How to build variable-length output records from fixed-length input records
- How to process multiple input files
- The requirements for using sorted input files
- Populating a direct access file with KBUILD

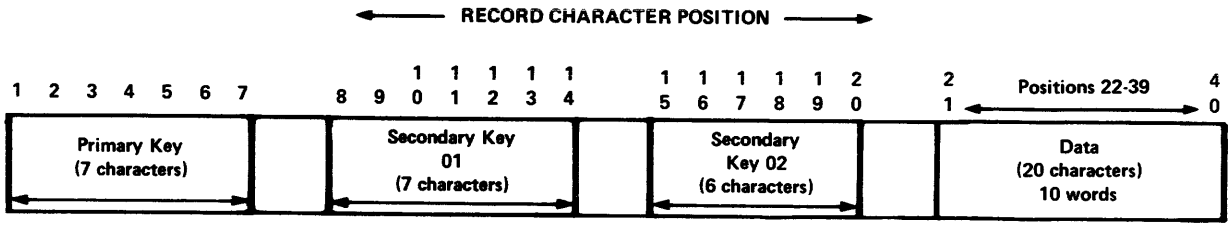


Figure 3-1a. Keys Resident in Data

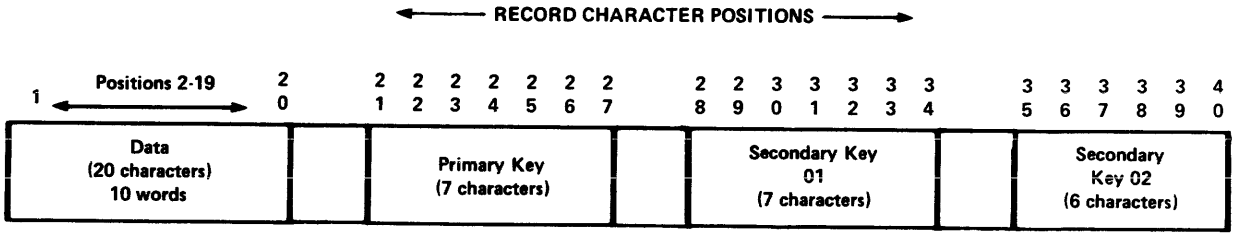


Figure 3-1b. Keys Not Resident in Data

Figure 3-1. Sample Record Layouts

Building a MIDAS File With Variable-Length Records

KBUILD can be used to process input files with fixed-length records for addition to MIDAS files that have variable-length records. There is a catch, however. In each record of the input file the user must indicate the number of words in that record which KBUILD should actually add to the data subfile. For ease of reference, let's call this piece of information the output record length. This record length can be specified as a bit string (an integer in binary form) or as an ASCII string. KBUILD asks you what form it's in so it can be properly converted. This number must appear at the same word position in each input file record. KBUILD asks whether the number is a bit string (B) or an ASCII string (A) during the dialog sequence, after it determines that your output file has variable-length records.

For example, word number 15 of each of these input file records is used to indicate the number of words that should be processed from the record:

```
2194GSpectrographics        117 Lyons Blvd, Jamaica Pln.
1002PFlora Portraits         11 Ramos Ave, Belleville
9411PStudio West             09 Arcade Ct, Pasadena
9402AArtistry Unltd.         10 Moorland St, Utica
0816SMorrow Paper Mills      12 Vista Point, Monadnock
2334PSeacoast Strippers      12 Seaspray Ln, Monterey
4056SMark-Burton             09 Granger Rd, Belmont
```

In this example, the output record length is in ASCII form, as the input file was created with the PRIMOS Editor (ED). Since it's not likely that the user wants the data transfer length to be included in the actual data written to the data subfile, so make this word number the last item in the input file record. It could also be put in the front of the record, but it would then become part of the data subfile record by default (and you probably wouldn't want it that way).

Requirements for Variable-Length Records: When you have a MIDAS file with variable-length records, KBUILD assumes you want to add variable-length records to it, although it's perfectly reasonable to add fixed-length records to such a file. However, KBUILD requires you to specify the input record length, because it must have fixed-length input records to process. Therefore, even though KBUILD can add records to a variable-length MIDAS file, it requires the input file to have fixed-length records. Once KBUILD has determined that the MIDAS file indeed has variable-length records, KBUILD prints out the following message:

```
THE OUTPUT FILE SELECTED CONTAINS VARIABLE LENGTH DATA RECORDS.
IS THE OUTPUT RECORD LENGTH SPECIFIED IN EACH INPUT RECORD
AN ASCII STRING OR A BINARY(INT*2) STRING?  (ENTER A OR B):
```

Your answer depends upon the format of the input file.

If the number is in ASCII format, and you entered "A" to the above prompt, KBUILD then asks:

```
ENTER STARTING CHARACTER POSITION IN INPUT RECORD:
ENTER ENDING CHARACTER POSITION IN INPUT RECORD:
```

Simply enter the starting and ending character positions of the output record length indicator in each input file record.

If the number is a binary string (INT*2 number), you must supply the word number at which the output record length indicator appears in the input file record:

```
ENTER STARTING WORD NUMBER IN INPUT RECORD:
```

Supply the word number in the input file record reserved for the output record length indicator. If the input file does not have a word number set aside for this purpose, KBUILD cannot process the file properly and will abort. You can specify fixed-length records if desired by simply making the output record length the same in each input record.

Note

The routines PRIBLD, SECBLD and BILD$R are still available for users who would rather build variable-length MIDAS files that way than with KBUILD. These routines also support direct access files. See Section 14 for information.

Multiple Input Files

KBUILD allows more than one input file to be processed during a single run. This allows you to add information from several data files, up to 100 of them, to a single MIDAS file.

Note

It is not possible to have more than one output file open at a time during a single execution of KBUILD.

If more than one input file exists, the filenames must all begin with the same letters and end in a two-digit number, beginning with any two-digit number you want. For example:

```
CUST01
CUST02
CUST03
```

Up to 100 input files can be processed at a time. The files must all exist in the same directory and must have exactly the same format and file type. It is also assumed that the key fields begin in the same record position in all of the input files.

Sorted Input Files

Input files can be sorted (in ascending order only) by primary and/or secondary key. If sorted by primary key, the data records and key entries are all added in primary key order. That is, the order of the entries in the data subfile will correspond to the order of the respective key entries in the primary index subfile. If the file contains secondary index entries for a subfile that allows duplicates, these duplicate keys will be added to the index subfile in the order in which they are read from the input file.

Pre-sorted index entries can only be added to an index that contains no entries. This is true for both primary and secondary index subfiles. There may be times when an index is for all purposes "empty" because it contains entries that point to deleted data subfile records. KBUILD can recognize that such an index is logically empty only if the primary index for that file is truly empty; this is because a file with an empty primary index doesn't have any data subfile records to be referenced, so any secondary index entries that exist are useless. When KBUILD notices that you are trying to build a non-empty secondary index from a sorted input file, it tells you so. MPACK must be run on this index to clean it out before you can add sorted input entries to it. Otherwise, if the primary index does contain entries, you must run KIDDEL to clean out the secondary index subfile you want to build before trying to add sorted entries to it.

Sort Requirements: The KBUILD dialog asks whether input files are sorted or unsorted; if the files are sorted, it asks if the primary key field has been sorted. KBUILD then asks if any secondary key fields have been used as sort keys. There are some restrictions on when a file can be properly called a "sorted" file:

- If the input file records have not been sorted by a primary or secondary key field, the file is unsorted.
- If there are several input files, they must all be sorted on the same field and all the sorted key values in the first file must be less than the sorted key values of the second file (and so on for as many files as you've got) in order to call the files sorted. This holds true for each sorted key field in the file. If either of these requirements is not met, the files must be declared unsorted.
- Furthermore, if an index in the MIDAS file to which the entries are being added already contains entries, do not declare the input files sorted by that key even if they are. If you do, KBUILD will catch this error and tell you about it. KBUILD cannot process an input file for building a MIDAS index that already contains entries if that input file is called "sorted."

Multiple Sort Keys: When a file is specified as "sorted," KBUILD asks whether this file (or files) is sorted by primary key. Regardless of

the user's answer to this question, KBUILD then asks which secondary key it has been sorted on. This prompt is repeated to allow the user to specify multiple index numbers. You should then indicate the numbers of any secondary key fields by which the input file has been sorted. If the file has not been sorted by a secondary key field, hit (CR) in response to this prompt. Do likewise when you've finished telling KBUILD which secondary keys were used in sorting the file.

Regardless of whether the input files are sorted or not, the data subfile entries are always stored in the order they are read in. Index entries are ultimately stored in sorted order, whether the user sorts them or not (it just saves time to pre-sort them). Duplicates are stored in the order read in, regardless of sorting.

Adding Secondary Index Entries Only

Most of this discussion on KBUILD has been based on the assumption that all primary and secondary key entries are being added along with data subfile entries. However, KBUILD is quite handy for populating a secondary index subfile when:

- You decide to make one of the fields in a MIDAS data record a secondary key. (You need more keys.)
- You didn't supply secondary key values supplied for all the data entries you added originally; thus, this index subfile is "sparse," meaning there is not a one-to-one correspondence between index entries and data subfile entries.

To accomplish this, you have two choices:

- Get the secondary index entries (key values) from the MIDAS data subfile records (i.e., make one of the fields in the data record a secondary key).
- Take the secondary index entries from an external input file. In this case, the primary key must be present in the input record.

Remember -- secondary index subfiles must be added to the template with CREATK before KBUILD can add entries to them. In other words, KBUILD can build the index subfiles as long as they've been properly defined, but KBUILD itself cannot define a new index subfile.

The KBUILD Dialog

To invoke the KBUILD utility, simply type KBUILD. Responses can be entered in upper- or lowercase, as is the case for all MIDAS utilities at Rev 17.6. The general KBUILD dialog is shown below, with prompts numbered for convenience.

Prompt	Response
1. SECONDARIES ONLY?	Enter Y[ES] or N[O]: YES: builds/adds entries to one or more secondary index subfiles. Dialog resumes at step 2. (Subfiles may or may not contain entries.) NO: adds data entries to data subfile by primary key. Entries are also added to primary index subfile and any secondary index subfiles as indicated. Dialog resumes at step 4.
2. USE MIDAS DATA?	Enter Y[ES] or N[O]: YES: existing data entries will be used as a source of values for a secondary index subfile(s). Do this when you've got existing records in the data subfile and you want to make fields from these records into secondary keys. Dialog continues with step 3. NO: all subfile and data subfile entries are taken from an input file (not a MIDAS file) that must contain primary key values too. Use this method when the MIDAS file being built does not contain any data entries. Dialog continues with step 4.
3. ENTER MIDAS FILENAME:	Enter pathname of MIDAS file from which KBUILD should extract the secondary key entries; asked only if you answered YES to prompt 2. Dialog skips to step 14.
4. ENTER INPUT FILENAME:	Enter pathname of input file to be processed by KBUILD. If using multiple files, simply enter the name of the one with the lowest sequence number.
5. ENTER INPUT RECORD LENGTH (WORDS):	Enter size of input file record in words.

6. INPUT FILE TYPE: Enter one of the KBUILD file type codes; see Table 3-2.

7. ENTER NUMBER OF FILES: Enter 1 for single files; for multiple input files, enter the total number of files.

8. ENTER OUTPUT FILENAME: Enter pathname of MIDAS file to which input file is being added.

(If the MIDAS output file has variable-length records, the next prompt is displayed. If the MIDAS file has fixed-length records, the dialog skips to step 13. For direct access files, see KBUILD and DIRECT ACCESS, below.)

9. THE OUTPUT FILE SELECTED CONTAINS VARIABLE LENGTH DATA RECORDS. IS THE OUTPUT RECORD LENGTH SPECIFIED IN EACH INPUT RECORD AN ASCII STRING OR A BINARY(INT*2) STRING? (ENTER A OR B):

In general (and this is a general statement, not a rule), if the input file was created with the Editor, COBOL, or RPG, and the output record length is in ASCII form, enter "A"; dialog continues with step 10. If the input file is BINARY or FTNBIN enter "B"; dialog continues with step 12.

10. ENTER STARTING CHARACTER POSITION IN INPUT RECORD: Enter the character position where the output record length specification begins. (Asked only if "A" specified above.)

11. ENTER ENDING CHARACTER POSITION IN INPUT RECORD: Enter character position that marks the end of the output record length specification. (Asked only if "A" was specified for prompt 9.) Dialog resumes with step 13.

12. ENTER STARTING WORD NUMBER IN INPUT RECORD: Enter the word number in the input record that specifies the output record length: for Binary (INT*2) representations only. (Asked only if "B" was specified in response to prompt 9.)

13. ENTER STARTING CHARACTER POSITION, PRIMARY KEY: Enter starting position of field in input record which contains primary key value. (Asked only if answer to initial KBUILD prompt was NO.)

14.	SECONDARY KEY NUMBER:	Enter number of secondary index subfile for which an entry is to be taken from the input file record. This and the next prompt are repeated until the user hits (CR).
15.	ENTER STARTING CHARACTER POSITION:	Enter character position in input record where this secondary key field begins.
16.	IS THE FILE SORTED?	Enter YES only if the index being built contains no entries and the input file or files are indeed sorted (and all in the same way); otherwise, enter NO, and the dialog resumes at step 19.

(next two prompts asked only if you answered YES to previous prompt)

17.	IS THE PRIMARY KEY SORTED:	Enter YES if input file was sorted by primary key field. File may be sorted additionally by a secondary key field. Enter NO if file was not sorted by primary key field. Not asked if building secondary index entries only.
18.	ENTER INDEX NUMBER OF SECONDARY SORT KEY:	If file was sorted on a field that corresponds to a secondary key, enter that key (index subfile) number. Prompt is repeated until you hit (CR).
19.	ENTER LOG/ERROR FILE NAME:	Enter name of file to be opened for recording errors and KBUILD statistics: see below. The filename should not be the name of an existing file. If it is, the existing file will be overwritten. Hit (CR) if you don't want the statistics recorded; they are still displayed at the terminal however.

2Ø. ENTER MILESTONE COUNT:	Enter interval (number of records) at which statistics should be displayed (and optionally recorded in a log/error file) during processing of the input file. If you enter Ø, milestones are printed for first and last records of input file only.

Logging Errors and Milestones

KBUILD automatically reports all errors and milestone statistics at the user's terminal. It also displays the name of the input file it is currently processing and tells you what part of the MIDAS file it is currently in. This data can be optionally recorded in a log/error file which the user indicates during the KBUILD dialog. If you don't want such a file to be created, simply supply a carriage return (CR) when KBUILD asks you for the log/error filename. The statistics will be displayed at the terminal but they won't be saved in a file.

Milestone Reporting: A milestone report consists of:

- The number of the record for which the milestone is being generated
- The current date and time
- The CPU time elapsed since KBUILD began processing this file
- The disk time used since KBUILD began processing
- The total disk and CPU time elapsed since the start of KBUILD's run
- The increment (of total time) elapsed since the last milestone was generated

If the input file is unsorted, KBUILD also tells you when it begins and ends a sort pass through each set of index entries.

Here is an example of a milestone report for a sample run of KBUILD, using an unsorted input file. The milestone count is 1:

```
BUILDING: DATA
DEFERRING: Ø, 1

PROCESSING FROM:  var.names
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM     DIFF
       Ø  Ø9-2Ø-8Ø  23:31:33     Ø.ØØØ      Ø.ØØØ      Ø.ØØØ     Ø.ØØØ
       1  Ø9-2Ø-8Ø  23:31:33     Ø.ØØ2      Ø.ØØØ      Ø.ØØ2     Ø.ØØ2
       2  Ø9-2Ø-8Ø  23:31:34     Ø.Ø19      Ø.ØØØ      Ø.Ø19     Ø.Ø16
       3  Ø9-2Ø-8Ø  23:31:34     Ø.Ø2Ø      Ø.ØØØ      Ø.Ø2Ø     Ø.ØØ1
       4  Ø9-2Ø-8Ø  23:31:36     Ø.Ø34      Ø.ØØØ      Ø.Ø34     Ø.Ø14
       5  Ø9-2Ø-8Ø  23:31:36     Ø.Ø35      Ø.ØØ1      Ø.Ø36     Ø.ØØ2
FIRST BUILD/DEFER PASS COMPLETE.
       5  Ø9-2Ø-8Ø  23:31:36     Ø.Ø36      Ø.ØØ1      Ø.Ø37     Ø.ØØ1

SORTING INDEX Ø
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM     DIFF
       Ø  Ø9-2Ø-8Ø  23:31:36     Ø.ØØØ      Ø.ØØØ      Ø.ØØØ     Ø.ØØØ
SORT COMPLETE
       5  Ø9-2Ø-8Ø  23:31:36     Ø.ØØ4      Ø.ØØØ      Ø.ØØ4     Ø.ØØ4

BUILDING INDEX Ø
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM     DIFF
       Ø  Ø9-2Ø-8Ø  23:31:36     Ø.ØØØ      Ø.ØØØ      Ø.ØØØ     Ø.ØØØ
       1  Ø9-2Ø-8Ø  23:31:36     Ø.ØØ1      Ø.ØØØ      Ø.ØØ1     Ø.ØØ1
       2  Ø9-2Ø-8Ø  23:31:36     Ø.ØØ1      Ø.ØØØ      Ø.ØØ1     Ø.ØØØ
       3  Ø9-2Ø-8Ø  23:31:36     Ø.ØØ2      Ø.ØØØ      Ø.ØØ2     Ø.ØØØ
       4  Ø9-2Ø-8Ø  23:31:36     Ø.ØØ2      Ø.ØØØ      Ø.ØØ2     Ø.ØØØ
       5  Ø9-2Ø-8Ø  23:31:36     Ø.ØØ2      Ø.ØØØ      Ø.ØØ2     Ø.ØØØ
INDEX Ø BUILT
       5  Ø9-2Ø-8Ø  23:31:37     Ø.ØØ3      Ø.ØØØ      Ø.ØØ3     Ø.ØØ1

SORTING INDEX  1
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM     DIFF
       Ø  Ø9-2Ø-8Ø  23:31:37     Ø.ØØØ      Ø.ØØØ      Ø.ØØØ     Ø.ØØØ
SORT COMPLETE
       5  Ø9-2Ø-8Ø  23:31:37     Ø.ØØ4      Ø.ØØØ      Ø.ØØ4     Ø.ØØ4

BUILDING INDEX   1
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM     DIFF
       Ø  Ø9-2Ø-8Ø  23:31:37     Ø.ØØØ      Ø.ØØØ      Ø.ØØØ     Ø.ØØØ
       1  Ø9-2Ø-8Ø  23:31:37     Ø.ØØ1      Ø.ØØ1      Ø.ØØ2     Ø.ØØ2
       2  Ø9-2Ø-8Ø  23:31:37     Ø.ØØ2      Ø.ØØ1      Ø.ØØ2     Ø.ØØØ
       3  Ø9-2Ø-8Ø  23:31:37     Ø.ØØ2      Ø.ØØ1      Ø.ØØ3     Ø.ØØØ
       4  Ø9-2Ø-8Ø  23:31:37     Ø.ØØ3      Ø.ØØ1      Ø.ØØ3     Ø.ØØØ
       5  Ø9-2Ø-8Ø  23:31:37     Ø.ØØ3      Ø.ØØ1      Ø.ØØ4     Ø.ØØØ
INDEX  1 BUILT
       5  Ø9-2Ø-8Ø  23:31:37     Ø.ØØ4      Ø.ØØ1      Ø.ØØ4     Ø.ØØ1

KBUILD COMPLETE.
```

The milestones were done for each record in the input file because there were so few of them. For larger files, set the milestone count to a more appropriate value depending on how concerned you are with resource usage.

If, for example, the milestone increment is 10 and there are 35 records in the file, KBUILD prints the words:

FIRST BUILD/DEFER PASS COMPLETE.

or

INDEX xx BUILT

before the milestone for record 35 is displayed. xx is the number of the index which was just completed. Following this message, the number 35 will be displayed, indicating that a total of 35 records were added to the MIDAS file.

Milestone Reports for Multiple Files: If there are multiple input files, the name of each successive input file is displayed above the record count column as each new input file is processed.

Error Reporting: KBUILD reports any error (both fatal and non-fatal) that it encounters during processing. Errors can be file handler errors or MIDAS errors. The type and number of the error encountered are printed, along with the record number that was being processed when this error occurred. Fatal errors are recorded in the log/error file just before KBUILD aborts.

KBUILD Examples

The examples shown below illustrate these features of KBUILD:

- Building a fixed-length record MIDAS file from unsorted input
- Building a fixed-length MIDAS file from sorted input
- Building a variable-length record MIDAS file from text input
- Building a secondary index from existing data subfile entries

Using Unsorted Input: In this example, an unsorted input file is used to build entries for the CUSTOMER file which has fixed-length records of 35 words. Input files are not required to have the same record length as that of the output (MIDAS) file. When added, if they are too long, they are truncated, and if they are too short, they are blank-padded to the correct length. The input file contains the following records:

```
2194GSpectrographics          NWOR
1002PFlora Portraits          NENY
9411PStudio West              WRCA
9402AArtistry Unltd.          WRCA
0816SMorrow Paper Mills       NENH
2334PSeacoast Strippers       WRCA
4056SMark-Burton              NEMA
```

The file was built using the dialog shown below:

```
OK, kbuild
[KBUILD rev 17.6]

SECONDARIES ONLY? no
ENTER INPUT FILENAME: names
ENTER INPUT RECORD LENGTH (WORDS): 35
INPUT FILE TYPE: t
ENTER NUMBER OF INPUT FILES: 1
ENTER OUTPUT FILENAME: customer
ENTER STARTING CHARACTER POSITION, PRIMARY KEY: 1
SECONDARY KEY NUMBER: 1
ENTER STARTING CHARACTER POSITION: 6
SECONDARY KEY NUMBER: 2
ENTER STARTING CHARACTER POSITION: 31
SECONDARY KEY NUMBER:  (CR)
IS FILE SORTED? n
ENTER LOG/ERROR FILE NAME: (CR)
ENTER MILESTONE COUNT: 1
```

```
BUILDING: DATA
DEFERRING: 0, 1, 2

PROCESSING FROM:  names
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM     DIFF
       0  09-23-80  10:28:50      0.000      0.000      0.000      0.000
       1  09-23-80  10:28:50      0.002      0.001      0.003      0.003
       2  09-23-80  10:28:50      0.002      0.001      0.003      0.001
       3  09-23-80  10:28:50      0.003      0.001      0.004      0.001
       4  09-23-80  10:28:50      0.003      0.001      0.005      0.001
       5  09-23-80  10:28:50      0.004      0.001      0.005      0.001
FIRST BUILD/DEFER PASS COMPLETE.
       5  09-23-80  10:28:51      0.004      0.001      0.006      0.001

SORTING INDEX 0
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM     DIFF
       0  09-23-80  10:28:51      0.000      0.000      0.000      0.000
SORT COMPLETE
       5  09-23-80  10:28:51      0.004      0.000      0.004      0.004

BUILDING INDEX 0
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM     DIFF
       0  09-23-80  10:28:51      0.000      0.000      0.000      0.000
       1  09-23-80  10:28:51      0.001      0.000      0.001      0.001
       2  09-23-80  10:28:51      0.001      0.000      0.001      0.000
       3  09-23-80  10:28:51      0.002      0.000      0.002      0.000
       4  09-23-80  10:28:51      0.002      0.000      0.002      0.000
       5  09-23-80  10:28:51      0.002      0.000      0.002      0.000
INDEX 0 BUILT
       5  09-23-80  10:28:51      0.003      0.000      0.003      0.001

SORTING INDEX  1
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM     DIFF
       0  09-23-80  10:28:52      0.000      0.000      0.000      0.000
SORT COMPLETE
       5  09-23-80  10:28:52      0.004      0.000      0.004      0.004

BUILDING INDEX  1
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM     DIFF
       0  09-23-80  10:28:52      0.000      0.000      0.000      0.000
       1  09-23-80  10:28:52      0.002      0.001      0.002      0.002
       2  09-23-80  10:28:52      0.002      0.001      0.003      0.000
       3  09-23-80  10:28:52      0.002      0.001      0.003      0.000
       4  09-23-80  10:28:52      0.003      0.001      0.004      0.000
       5  09-23-80  10:28:52      0.003      0.001      0.004      0.000
INDEX  1 BUILT
       5  09-23-80  10:28:52      0.004      0.001      0.005      0.001

SORTING INDEX  2
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM     DIFF
       0  09-23-80  10:28:52      0.000      0.000      0.000      0.000
SORT COMPLETE
       5  09-23-80  10:28:53      0.004      0.000      0.004      0.004
```

```
BUILDING INDEX   2
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM     DIFF
       Ø  Ø9-23-8Ø  1Ø:28:53     Ø.ØØØ      Ø.ØØØ      Ø.ØØØ     Ø.ØØØ
       1  Ø9-23-8Ø  1Ø:28:53     Ø.ØØ1      Ø.ØØ1      Ø.ØØ3     Ø.ØØ3
       2  Ø9-23-8Ø  1Ø:28:53     Ø.ØØ2      Ø.ØØ1      Ø.ØØ3     Ø.ØØØ
       3  Ø9-23-8Ø  1Ø:28:53     Ø.ØØ2      Ø.ØØ1      Ø.ØØ3     Ø.ØØØ
       4  Ø9-23-8Ø  1Ø:28:53     Ø.ØØ3      Ø.ØØ1      Ø.ØØ4     Ø.ØØØ
       5  Ø9-23-8Ø  1Ø:28:53     Ø.ØØ3      Ø.ØØ1      Ø.ØØ4     Ø.ØØØ
INDEX  2 BUILT
       5  Ø9-23-8Ø  1Ø:28:53     Ø.ØØ4      Ø.ØØ1      Ø.ØØ5     Ø.ØØ1

KBUILD COMPLETE.
```

Using Sorted Input: If the CUSTOMER file is built from sorted input records, the build is faster because KBUILD doesn't have to sort the records itself. This is the sorted input file (sorted on primary key):

```
Ø816SMorrow Paper Mills       NENH
1ØØ2PFlora Portraits          NENY
2194GSpectrographics          NWOR
94Ø2AArtistry Unltd.          WRCA
9411PStudio West              WRCA
```

The KBUILD-user dialog used to build this file is:

```
OK, kbuild
[KBUILD rev 17.6]

SECONDARIES ONLY? no
ENTER INPUT FILENAME: names.sort
ENTER INPUT RECORD LENGTH (WORDS): 17
INPUT FILE TYPE: text
ENTER NUMBER OF INPUT FILES: 1
ENTER OUTPUT FILENAME: customer
ENTER STARTING CHARACTER POSITION, PRIMARY KEY: 1
SECONDARY KEY NUMBER: 1
ENTER STARTING CHARACTER POSITION: 6
SECONDARY KEY NUMBER: 2
ENTER STARTING CHARACTER POSITION: 31
SECONDARY KEY NUMBER: (CR)
IS FILE SORTED? yes
IS THE PRIMARY KEY SORTED? yes
ENTER INDEX NUMBER OF SECONDARY SORT KEY:  (CR)
ENTER LOG/ERROR FILE NAME: (CR)
ENTER MILESTONE COUNT: 1
```

```
BUILDING: DATA, Ø
DEFERRING: 1, 2

PROCESSING FROM:  names.sort
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM     DIFF
       Ø  Ø9-22-8Ø  15:28:45     Ø.ØØØ      Ø.ØØØ      Ø.ØØØ      Ø.ØØØ
       1  Ø9-22-8Ø  15:28:45     Ø.ØØ2      Ø.ØØ3      Ø.ØØ4      Ø.ØØ4
       2  Ø9-22-8Ø  15:28:45     Ø.ØØ2      Ø.ØØ3      Ø.ØØ5      Ø.ØØ1
       3  Ø9-22-8Ø  15:28:45     Ø.ØØ3      Ø.ØØ3      Ø.ØØ5      Ø.ØØ1
       4  Ø9-22-8Ø  15:28:45     Ø.ØØ3      Ø.ØØ3      Ø.ØØ6      Ø.ØØ1
       5  Ø9-22-8Ø  15:28:45     Ø.ØØ4      Ø.ØØ3      Ø.ØØ6      Ø.ØØ1
FIRST BUILD/DEFER PASS COMPLETE.
       5  Ø9-22-8Ø  15:28:46     Ø.ØØ4      Ø.ØØ3      Ø.ØØ7      Ø.ØØ1

SORTING INDEX  1
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM     DIFF
       Ø  Ø9-22-8Ø  15:28:46     Ø.ØØØ      Ø.ØØØ      Ø.ØØØ      Ø.ØØØ
SORT COMPLETE
       5  Ø9-22-8Ø  15:28:47     Ø.ØØ4      Ø.ØØ5      Ø.ØØ9      Ø.ØØ9

BUILDING INDEX   1
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM     DIFF
       Ø  Ø9-22-8Ø  15:28:47     Ø.ØØØ      Ø.ØØØ      Ø.ØØØ      Ø.ØØØ
       1  Ø9-22-8Ø  15:28:47     Ø.ØØ2      Ø.ØØ2      Ø.ØØ4      Ø.ØØ4
       2  Ø9-22-8Ø  15:28:47     Ø.ØØ2      Ø.ØØ3      Ø.ØØ5      Ø.ØØ1
       3  Ø9-22-8Ø  15:28:47     Ø.ØØ3      Ø.ØØ3      Ø.ØØ6      Ø.ØØØ
       4  Ø9-22-8Ø  15:28:47     Ø.ØØ3      Ø.ØØ3      Ø.ØØ6      Ø.ØØØ
       5  Ø9-22-8Ø  15:28:47     Ø.ØØ3      Ø.ØØ3      Ø.ØØ7      Ø.ØØØ
INDEX  1 BUILT
       5  Ø9-22-8Ø  15:28:47     Ø.ØØ4      Ø.ØØ3      Ø.ØØ7      Ø.ØØ1

SORTING INDEX  2
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM     DIFF
       Ø  Ø9-22-8Ø  15:28:47     Ø.ØØØ      Ø.ØØØ      Ø.ØØØ      Ø.ØØØ
SORT COMPLETE
       5  Ø9-22-8Ø  15:28:48     Ø.ØØ3      Ø.ØØØ      Ø.ØØ3      Ø.ØØ3

BUILDING INDEX   2
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM     DIFF
       Ø  Ø9-22-8Ø  15:28:48     Ø.ØØØ      Ø.ØØØ      Ø.ØØØ      Ø.ØØØ
       1  Ø9-22-8Ø  15:28:49     Ø.ØØ2      Ø.ØØ2      Ø.ØØ3      Ø.ØØ3
       2  Ø9-22-8Ø  15:28:49     Ø.ØØ2      Ø.ØØ2      Ø.ØØ4      Ø.ØØ1
       3  Ø9-22-8Ø  15:28:49     Ø.ØØ2      Ø.ØØ2      Ø.ØØ4      Ø.ØØØ
       4  Ø9-22-8Ø  15:28:49     Ø.ØØ3      Ø.ØØ2      Ø.ØØ4      Ø.ØØØ
       5  Ø9-22-8Ø  15:28:49     Ø.ØØ3      Ø.ØØ2      Ø.ØØ5      Ø.ØØ1
INDEX  2 BUILT
       5  Ø9-22-8Ø  15:28:49     Ø.ØØ4      Ø.ØØ2      Ø.ØØ6      Ø.ØØ1
KBUILD COMPLETE.
```

Note that the primary index entries were not sorted during this run of KBUILD because we told it that they'd already been sorted. However, the secondary index entries were sorted by KBUILD as indicated by the explanatory messages.

Building Variable-length Records: MIDAS files with variable-length records can be built by KBUILD, but the user must supply KBUILD with the length of each data record to be written to the output file. This example shows the prompts needed to create a keyed index file with variable-length records, the input file with the output record length indicated in each record and the user/KBUILD dialog needed to build this MIDAS file.

```
OK, creatk
[CREATK rev 17.6]

MINIMUM OPTIONS? yes

FILE NAME? varcust
NEW FILE? yes
DIRECT ACCESS? no

DATA SUBFILE QUESTIONS

PRIMARY KEY TYPE: a
PRIMARY KEY SIZE = : b 5
DATA SIZE = : (CR)

SECONDARY INDEX

INDEX NO.? 1

DUPLICATE KEYS PERMITTED? yes

KEY TYPE: a
KEY SIZE = : b 25
SECONDARY DATA SIZE = : (CR)

INDEX NO.? 2

DUPLICATE KEYS PERMITTED? yes

KEY TYPE: a
KEY SIZE = : b 4
SECONDARY DATA SIZE = :  (CR)

INDEX NO.?  (CR)

SETTING FILE LOCK TO N READERS AND N WRITERS
```

The input file used in building the MIDAS file is called VAR.NAMES. Although each output record is different in length, KBUILD needs to know the input data record length. In this case, it is 22 words. Each

record in the file contains a number that indicates the output record length for that particular record. This length tells KBUILD how many words of the input record to write to the data subfile. The output record length begins in character position 30 and ends in character position 31 of each record. Observe that the address information will not be written to the file as part of the data subfile record. At a later time however, another index could be created and the address field corresponding to each record in the data subfile could be added to the new index subfile.

```
2194GSpectrographics          11  7 Lyons Blvd, Jamaica Pln.
1002PFlora Portraits          11  Ramos Ave, Belleville
9411PStudio West              09  Arcade Ct, Pasadena
9402AArtistry Unltd.          10  Moorland St, Utica
0816SMorrow Paper Mills       12  Vista Point, Monadnock
2334PSeacoast Strippers       12  Seaspray Ln, Monterey
4056SMark-Burton              09  Granger Rd, Belmont
```

Since the output record length is in ASCII form, we tell KBUILD what character positions it begins and ends in, that is, 30 and 31. If the number were in binary (INTEGER*2) form, we'd indicate the word number that the entry number begins in. The KBUILD dialog and user responses for the above example are as follows:

```
OK, kbuild
[KBUILD rev 17.6]

SECONDARIES ONLY? no
ENTER INPUT FILENAME: var.names
ENTER INPUT RECORD LENGTH (WORDS): 22
INPUT FILE TYPE: t
ENTER NUMBER OF INPUT FILES: 1
ENTER OUTPUT FILENAME: varcust
THE OUTPUT FILE SELECTED CONTAINS VARIABLE LENGTH DATA RECORDS.
IS THE OUTPUT RECORD LENGTH SPECIFIED IN EACH INPUT RECORD
AN ASCII STRING OR A BINARY (INT*2) STRING? (ENTER A OR B): a
ENTER STARTING CHARACTER POSITION IN INPUT RECORD: 30
ENTER ENDING CHARACTER POSITION IN INPUT RECORD: 31
ENTER STARTING CHARACTER POSITION, PRIMARY KEY: 1
SECONDARY KEY NUMBER: 1
ENTER STARTING CHARACTER POSITION: 6
SECONDARY KEY NUMBER: (CR)
IS FILE SORTED? no
ENTER LOG/ERROR FILE NAME:  (CR)
ENTER MILESTONE COUNT: 1
```

```
BUILDING: DATA
DEFERRING: Ø, 1

PROCESSING FROM:  var.names
 COUNT      DATE      TIME      CPU MIN    DISK MIN    TOTAL TM      DIFF
       Ø  Ø9-3Ø-8Ø  23:55:13      Ø.ØØØ       Ø.ØØØ       Ø.ØØØ       Ø.ØØØ
       1  Ø9-3Ø-8Ø  23:55:13      Ø.ØØ2       Ø.ØØØ       Ø.ØØ2       Ø.ØØ2
       2  Ø9-3Ø-8Ø  23:55:13      Ø.ØØ2       Ø.ØØØ       Ø.ØØ2       Ø.ØØ1
       3  Ø9-3Ø-8Ø  23:55:14      Ø.ØØ3       Ø.ØØØ       Ø.ØØ3       Ø.ØØ1
       4  Ø9-3Ø-8Ø  23:55:14      Ø.ØØ3       Ø.ØØØ       Ø.ØØ3       Ø.ØØ1
       5  Ø9-3Ø-8Ø  23:55:14      Ø.ØØ4       Ø.ØØØ       Ø.ØØ4       Ø.ØØ1
FIRST BUILD/DEFER PASS COMPLETE.
       5  Ø9-3Ø-8Ø  23:55:15      Ø.ØØ4       Ø.ØØØ       Ø.ØØ4       Ø.ØØ1

SORTING INDEX Ø
 COUNT      DATE      TIME      CPU MIN    DISK MIN    TOTAL TM      DIFF
       Ø  Ø9-3Ø-8Ø  23:55:15      Ø.ØØØ       Ø.ØØØ       Ø.ØØØ       Ø.ØØØ
SORT COMPLETE
       5  Ø9-3Ø-8Ø  23:55:16      Ø.ØØ4       Ø.ØØØ       Ø.ØØ4       Ø.ØØ4

BUILDING INDEX Ø
 COUNT      DATE      TIME      CPU MIN    DISK MIN    TOTAL TM      DIFF
       Ø  Ø9-3Ø-8Ø  23:55:16      Ø.ØØØ       Ø.ØØØ       Ø.ØØØ       Ø.ØØØ
       1  Ø9-3Ø-8Ø  23:55:17      Ø.ØØ1       Ø.ØØØ       Ø.ØØ1       Ø.ØØ1
       2  Ø9-3Ø-8Ø  23:55:17      Ø.ØØ1       Ø.ØØØ       Ø.ØØ1       Ø.ØØØ
       3  Ø9-3Ø-8Ø  23:55:17      Ø.ØØ2       Ø.ØØØ       Ø.ØØ2       Ø.ØØØ
       4  Ø9-3Ø-8Ø  23:55:17      Ø.ØØ2       Ø.ØØØ       Ø.ØØ2       Ø.ØØØ
       5  Ø9-3Ø-8Ø  23:55:17      Ø.ØØ2       Ø.ØØØ       Ø.ØØ2       Ø.ØØØ
INDEX Ø BUILT
       5  Ø9-3Ø-8Ø  23:55:18      Ø.ØØ3       Ø.ØØØ       Ø.ØØ3       Ø.ØØ1

SORTING INDEX  1
 COUNT      DATE      TIME      CPU MIN    DISK MIN    TOTAL TM      DIFF
       Ø  Ø9-3Ø-8Ø  23:55:18      Ø.ØØØ       Ø.ØØØ       Ø.ØØØ       Ø.ØØØ
SORT COMPLETE
       5  Ø9-3Ø-8Ø  23:55:2Ø      Ø.ØØ4       Ø.ØØØ       Ø.ØØ4       Ø.ØØ4

BUILDING INDEX   1
 COUNT      DATE      TIME      CPU MIN    DISK MIN    TOTAL TM      DIFF
       Ø  Ø9-3Ø-8Ø  23:55:2Ø      Ø.ØØØ       Ø.ØØØ       Ø.ØØØ       Ø.ØØØ
       1  Ø9-3Ø-8Ø  23:55:2Ø      Ø.ØØ2       Ø.ØØØ       Ø.ØØ2       Ø.ØØ2
       2  Ø9-3Ø-8Ø  23:55:21      Ø.ØØ2       Ø.ØØØ       Ø.ØØ2       Ø.ØØØ
       3  Ø9-3Ø-8Ø  23:55:21      Ø.ØØ3       Ø.ØØØ       Ø.ØØ3       Ø.ØØØ
       4  Ø9-3Ø-8Ø  23:55:21      Ø.ØØ3       Ø.ØØØ       Ø.ØØ3       Ø.ØØØ
       5  Ø9-3Ø-8Ø  23:55:21      Ø.ØØ3       Ø.ØØØ       Ø.ØØ3       Ø.ØØØ
INDEX  1 BUILT
       5  Ø9-3Ø-8Ø  23:55:21      Ø.ØØ4       Ø.ØØØ       Ø.ØØ4       Ø.ØØ1

KBUILD COMPLETE.
```

Building a Secondary Index From MIDAS Data: Suppose we only built the primary and one secondary index during the KBUILD as shown in the first example. At a later time we decide to add another secondary index to the file so that we can use the region code information as a search key. Since the information is already present in the data subfile record (we wrote the entire input record to the data subfile), we can tell KBUILD to take the secondary index entries from the data subfile record and add them to secondary index subfile 2. This example shows how. Keep in mind that we're working with a MIDAS file that already contains data entries.

To add a new secondary index to a file, use the ADD option of CREATK (see Section 12 for details):

```
OK, creatk

MINIMUM OPTIONS? y

FILE NAME? customer
NEW FILE? n

FUNCTION? add

INDEX NO.? 2

DUPLICATE KEYS PERMITTED? yes

KEY TYPE: a
KEY SIZE = : b 4
SECONDARY DATA SIZE = : (CR)


INDEX NO.? (CR)

FUNCTION? q
```

Entries were added to the new secondary index by providing KBUILD with the information as shown in this sample session:

```
OK, kbuild
[KBUILD rev 17.6]

SECONDARIES ONLY? yes
USE MIDAS DATA ENTRIES? yes

ENTER MIDAS FILENAME: customer
SECONDARY KEY NUMBER: 2
ENTER STARTING CHARACTER POSITION: 31
SECONDARY KEY NUMBER:  (CR)
IS FILE SORTED? n
ENTER LOG/ERROR FILE NAME: (CR)
ENTER MILESTONE COUNT: 1
```

```
DEFERRING: 2

PROCESSING FROM:  customer
 COUNT      DATE       TIME      CPU MIN    DISK MIN    TOTAL TM      DIFF
       Ø  1Ø-Ø1-8Ø  11:36:15       Ø.ØØØ       Ø.ØØØ       Ø.ØØØ       Ø.ØØØ
       1  1Ø-Ø1-8Ø  11:36:16       Ø.ØØ1       Ø.ØØ1       Ø.ØØ2       Ø.ØØ2
       2  1Ø-Ø1-8Ø  11:36:16       Ø.ØØ2       Ø.ØØ1       Ø.ØØ3       Ø.ØØ1
       3  1Ø-Ø1-8Ø  11:36:16       Ø.ØØ2       Ø.ØØ1       Ø.ØØ3       Ø.ØØØ
       4  1Ø-Ø1-8Ø  11:36:16       Ø.ØØ3       Ø.ØØ1       Ø.ØØ4       Ø.ØØ1
       5  1Ø-Ø1-8Ø  11:36:16       Ø.ØØ3       Ø.ØØ1       Ø.ØØ4       Ø.ØØØ
FIRST BUILD/DEFER PASS COMPLETE.
       5  1Ø-Ø1-8Ø  11:36:16       Ø.ØØ4       Ø.ØØ1       Ø.ØØ5       Ø.ØØ1

SORTING INDEX  2
 COUNT      DATE       TIME      CPU MIN    DISK MIN    TOTAL TM      DIFF
       Ø  1Ø-Ø1-8Ø  11:36:16       Ø.ØØØ       Ø.ØØØ       Ø.ØØØ       Ø.ØØØ
SORT COMPLETE
       5  1Ø-Ø1-8Ø  11:36:17       Ø.ØØ3       Ø.ØØØ       Ø.ØØ3       Ø.ØØ3

BUILDING INDEX   2
 COUNT      DATE       TIME      CPU MIN    DISK MIN    TOTAL TM      DIFF
       Ø  1Ø-Ø1-8Ø  11:36:17       Ø.ØØØ       Ø.ØØØ       Ø.ØØØ       Ø.ØØØ
       1  1Ø-Ø1-8Ø  11:36:17       Ø.ØØ2       Ø.ØØ1       Ø.ØØ3       Ø.ØØ3
       2  1Ø-Ø1-8Ø  11:36:17       Ø.ØØ2       Ø.ØØ1       Ø.ØØ3       Ø.ØØØ
       3  1Ø-Ø1-8Ø  11:36:18       Ø.ØØ2       Ø.ØØ1       Ø.ØØ4       Ø.ØØ1
       4  1Ø-Ø1-8Ø  11:36:18       Ø.ØØ3       Ø.ØØ1       Ø.ØØ4       Ø.ØØØ
       5  1Ø-Ø1-8Ø  11:36:18       Ø.ØØ3       Ø.ØØ1       Ø.ØØ4       Ø.ØØØ
INDEX  2 BUILT
       5  1Ø-Ø1-8Ø  11:36:18       Ø.ØØ4       Ø.ØØ1       Ø.ØØ5       Ø.ØØ1

KBUILD COMPLETE.
```

The MIDAS file now contains five entries in each of the indexes and the data subfile. (You can use CREATK to verify this.)

KBUILD AND DIRECT ACCESS

The remainder of this section is meant only for users with direct access MIDAS files. Direct access MIDAS files are called RELATIVE files in COBOL. and DIRECT files in RPG.

Building Direct Access Files

Practically everything said above about building keyed-index MIDAS files applies to direct access MIDAS files. The only major differences are:

- A record number must be supplied for each record -- the data type must be a REAL*4 (floating-point) number in the form of a bit string, or it can be an ASCII string.
- The record number must be placed at the same character position in each input record.
- In COBOL files, the record number must be the primary key.
- In non-COBOL files, a primary key can be supplied in addition to a record number (the record number doesn't have to be the primary key). A direct access file can have up to 999,999 entries.

KBUILD Dialog Requirements

When KBUILD determines that the MIDAS output file is a direct access file, it prints the following message:

```
IS THE ENTRY NUMBER SPECIFIED IN EACH INPUT RECORD
AN ASCII  STRING  OR A BINARY (REAL*4) STRING?  (ENTER A OR B):
```

KBUILD then prompts for the beginning and ending character positions of the record number if it is an ASCII string. If the number is specified as a single-precision floating-point bit string, KBUILD asks for the word number (not character position) at which the number begins in the record. If the word number is beyond the logical end of the record which you specified earlier in the dialog, KBUILD will warn you about it. See Section 11 for information on direct access (RELATIVE) files in COBOL. For more information on direct access files in RPG, see Section 1Ø.

Direct Access Example: A direct access file can be built with KBUILD, providing that you include record entry numbers for each record in the input file. Numbers can be written in ASCII or binary (floating point) form and should match the key type specification for the primary key if the primary key was defined as the record number, which is always the case in COBOL and RPG.

For example, this CREATK session sets up a direct access file with a

primary key declared as a 48-bit bit string. (This means it can be treated as a RELATIVE file in COBOL, because the primary key will be the record number.)

```
OK, creatk
[CREATK rev 17.6]

MINIMUM OPTIONS? yes

FILE NAME? dacust
NEW FILE? yes
DIRECT ACCESS? yes

DATA SUBFILE QUESTIONS

PRIMARY KEY TYPE: b
PRIMARY KEY SIZE = : b 48
DATA SIZE = : 22
NUMBER OF ENTRIES TO ALLOCATE? 10

SECONDARY INDEX

INDEX NO.? (CR)

SETTING FILE LOCK TO N READERS AND N WRITERS
```

The input file used to build the direct access file created above is shown below. Because we didn't want the record number to be part of the data subfile record, we put it after the fields that we did want included in the data subfile record. Thus the direct access entry number appears in positions 37 - 42 of the input file record. The numbers, like the entire input file, are written in ASCII format. COBOL programers might want to do something similar when building a RELATIVE file with KBUILD. The file used as input to KBUILD contains these records:

```
2194GSpectrographics          NWOR  000001
1002PFlora Portraits          NENY  000002
9411PStudio West              WRCA  000003
9402AArtistry Unltd.          WRCA  000004
0816SMorrow Paper Mills       NENH  000005
2334PSeacoast Strippers       WRCA  000006
4056SMark-Burton              NEMA  000007
```

This file was processed by KBUILD in the sample session shown here:

```
OK, kbuild
[KBUILD rev 17.6]

SECONDARIES ONLY? no
ENTER INPUT FILENAME: da.names
ENTER INPUT RECORD LENGTH (WORDS): 21
INPUT FILE TYPE: t
ENTER NUMBER OF INPUT FILES: 1
ENTER OUTPUT FILENAME: dacust
THE OUTPUT FILE SELECTED IS A DIRECT ACCESS FILE.
IS THE ENTRY NUMBER SPECIFIED IN EACH INPUT RECORD
AN ASCII STRING OR A BINARY (REAL*4) STRING? (ENTER A OR B): a
ENTER STARTING CHARACTER POSITION IN INPUT RECORD: 37
ENTER ENDING CHARACTER POSITION IN INPUT RECORD: 42
ENTER STARTING CHARACTER POSITION, PRIMARY KEY: 37
SECONDARY KEY NUMBER: (CR)
IS FILE SORTED? n
ENTER LOG/ERROR FILE NAME: (CR)
ENTER MILESTONE COUNT: 1

BUILDING: DATA
DEFERRING: Ø

PROCESSING FROM:  da.names.new
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM     DIFF
       Ø  10-20-8Ø  23:59:5Ø      Ø.ØØØ      Ø.ØØØ      Ø.ØØØ      Ø.ØØØ
       1  10-20-8Ø  23:59:5Ø      Ø.ØØ2      Ø.ØØØ      Ø.ØØ2      Ø.ØØ2
       2  10-20-8Ø  23:59:5Ø      Ø.ØØ2      Ø.ØØØ      Ø.ØØ2      Ø.ØØ1
       3  10-20-8Ø  23:59:5Ø      Ø.ØØ3      Ø.ØØØ      Ø.ØØ3      Ø.ØØ1
       4  10-20-8Ø  23:59:5Ø      Ø.ØØ3      Ø.ØØØ      Ø.ØØ3      Ø.ØØ1
       5  10-20-8Ø  23:59:5Ø      Ø.ØØ4      Ø.ØØØ      Ø.ØØ4      Ø.ØØ1
       6  10-20-8Ø  23:59:5Ø      Ø.ØØ5      Ø.ØØØ      Ø.ØØ5      Ø.ØØ1
FIRST BUILD/DEFER PASS COMPLETE.
       6  10-20-8Ø  23:59:5Ø      Ø.ØØ5      Ø.ØØØ      Ø.ØØ5      Ø.ØØ1

SORTING INDEX Ø
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM     DIFF
       Ø  10-20-8Ø  23:59:5Ø      Ø.ØØØ      Ø.ØØØ      Ø.ØØØ      Ø.ØØØ
SORT COMPLETE
       6  10-20-8Ø  23:59:51      Ø.ØØ4      Ø.ØØØ      Ø.ØØ4      Ø.ØØ4
```

```
BUILDING INDEX Ø
 COUNT      DATE      TIME      CPU MIN   DISK MIN   TOTAL TM      DIFF
       Ø  1Ø-2Ø-8Ø  23:59:51      Ø.ØØØ      Ø.ØØØ      Ø.ØØØ      Ø.ØØØ
       1  1Ø-2Ø-8Ø  23:59:51      Ø.ØØ1      Ø.ØØØ      Ø.ØØ1      Ø.ØØ1
       2  1Ø-2Ø-8Ø  23:59:51      Ø.ØØ1      Ø.ØØØ      Ø.ØØ1      Ø.ØØ1
       3  1Ø-2Ø-8Ø  23:59:51      Ø.ØØ2      Ø.ØØØ      Ø.ØØ2      Ø.ØØØ
       4  1Ø-2Ø-8Ø  23:59:51      Ø.ØØ2      Ø.ØØØ      Ø.ØØ2      Ø.ØØØ
       5  1Ø-2Ø-8Ø  23:59:51      Ø.ØØ2      Ø.ØØØ      Ø.ØØ2      Ø.ØØØ
       6  1Ø-2Ø-8Ø  23:59:51      Ø.ØØ3      Ø.ØØØ      Ø.ØØ3      Ø.ØØØ
INDEX Ø BUILT
       6  1Ø-2Ø-8Ø  23:59:51      Ø.ØØ4      Ø.ØØØ      Ø.ØØ4      Ø.ØØ1

KBUILD COMPLETE.
```

KBUILD ERROR MESSAGES

In addition to the messages that are listed in Appendix A [redacted] KBUILD will display one of the following messages during run-time if it gets into trouble. Some errors are fatal: these are always evidenced by the fact that KBUILD aborts after it reports them. In some "fatal" errors files may be damaged, but in most cases, they are still useable, as indicated in the error message explanations that follow.

▶ INVALID DIRECT ACCESS ENTRY NUMBER -- RECORD NOT ADDED

The user-supplied direct access record number is an ASCII string, but is not legitimate if it contains non-numeric characters. Also, the entry number may be less than or equal to Ø, may not be a whole number or may exceed number of records allocated. (Non-fatal)

▶ INVALID OUTPUT DATA RECORD LENGTH -- RECORD NOT ADDED

The output record length is an invalid ASCII string -- that is, it contains non-numeric characters. Also, the size specified might exceed the input record size. (Non-fatal)

▶ THIS INDEX IS NOT EMPTY. EITHER ZERO THE INDEX OR DO NOT SPECIFY THIS KEY AS SORTED.

KBUILD cannot add sorted data entries to any index subfile that already contains entries. (Non-fatal)

▶ INDEX BLOCK SIZE GREATER THAN MAXIMUM DEFAULT

The value of RECLNT in KPARAM used in building the KBUILD utility is smaller than the RECLNT value used when the file was created with CREATK. (Fatal)

▶ UNABLE TO REACH BOTTOM INDEX LEVEL

Couldn't locate last level index block -- file is damaged. (Fatal)

▶ CAN'T FIND PRIMARY KEY IN INDEX -- RECORD NOT ADDED

Occurs when adding secondary index entries. The primary key value supplied by the user was not found in the primary index. (Non-fatal)

▶ INDEX Ø: INVALID KEY -- RECORD NOT ADDED

Could occur if the input file is sorted and an entry was out of order, or if a duplicate key value appeared for an index that doesn't permit duplicates. (Non-fatal)

▶ INDEX Ø FULL -- INPUT TERMINATED

If the maximum number of entries in primary index is exceeded, KBUILD aborts. (Fatal -- but file still okay)

▶ INDEX index-no FULL -- NO MORE ENTRIES WILL BE ADDED TO IT

Same as above, but occurs during building of a secondary index. Building of other indexes continues. (Fatal -- but file still okay)

▶ INDEX Ø FULL — REMAINING RECORDS WILL BE DELETED

Data records are added to the subfile first, in the order read in from the input file. Then the primary index entries are added, in sorted order, to point to them. KBUILD ran out of room in the primary index when trying to add entries to point to those already in the data subfile and is forced to set the delete bit on in data subfile entries whose primary keys will not fit in the primary index. (Fatal -- but file still okay)

▶ INDEX{ Ø: / index-no: } KEY SEQUENCE ERROR — RECORD NOT ADDED

A duplicate value was discovered for the primary key or for a secondary key that doesn't allow duplicates. (Non-fatal)

SECTION 4

DELETING A MIDAS FILE (KIDDEL)

INTRODUCTION

This section covers the KIDDEL utility which is the fastest method of destroying or zeroing (removing entries from) part or all of an existing MIDAS file.

THE KIDDEL UTILITY

The MIDAS KIDDEL utility performs the following functions:

- Deletes an entire MIDAS file, including all index subfiles and data subfiles
- Deletes one or more secondary index subfiles (completely)
- Deletes "junk" (work files, etc.) left over from an aborted MPACK run (see Section 12)
- Zeroes out (initializes) one or more secondary index subfiles (removes all entries therefrom)
- Zeroes out, or initializes, all entries in the primary and secondary index subfiles, and all entries in the data subfile, if the file is a direct access MIDAS file

Zero vs. Delete

The KIDDEL dialog asks if you want to "delete" or "zero" a file's indexes. The difference between delete and zero is that delete gets rid of an entire index subfile, whereas "zero" only deletes the entries and unused space in an index subfile. An initialized or "zeroed out" file looks exactly like the initial template created for the file with CREATK.

The KIDDEL Dialog

The KIDDEL dialog shows how to respond to each prompt (prompts are numbered for convenience):

Prompt	Response
1. FILE NAME?	Enter pathname of MIDAS file to be KIDDELed.
2. DELETE INDEXES:	Enter one of the following:
	List of secondary indexes: numbers of secondary index subfiles to be deleted. Use optional commas between numbers.
	ALL: kills entire file, deletes it from the UFD it resides in, and returns you to PRIMOS.
	JUNK: deletes residual "garbage" left after an aborted MPACK operation.
	NO[NE]: you want no index subfiles deleted, but instead want to zero one or more of them. Dialog continues at step 3.
3. ZERO INDEXES:	Asked only if you entered "NONE" to above prompt. You may enter one of the following:
	List of indexes: the numbers of secondary index subfiles whose entries are to be deleted.
	ALL: zeroes all index subfiles and the data subfile. If a direct access file, the file is reinitialized.
	NONE: returns you without action to PRIMOS.

Note

It is not legal to enter "Ø" (primary index) in response to either prompt 2 or 3. If you want to delete or zero the primary index, you should use the "ALL" response.

KIDDEL Example

This example uses a version of the CUSTOMER file (created in Section 2) which has been accessed by both PL/I and BASIC/VM programs. Several records have been deleted from the file by primary key, and the secondary key entries which referenced these records still remain in the secondary index subfiles. KIDDEL is first used to remove entries

from the secondary indexes in the file. It is then used to zero all the index subfiles and finally, to delete one of the secondary index subfiles.

The "usage" option of CREATK is used to find out how many entries are in the index subfiles before and after KIDDEL is run. USAGE (abbreviated "u") was briefly mentioned in Section 2; See Section 12 for more information on this and other options.

Comments have been appended for explanatory purposes.

```
OK,                                  Get rid of entries in secondaries
OK, kiddel
[KIDDEL rev 17.6]

FILE NAME? customer

DELETE INDEXES: none

ZERO INDEXES: 1,2
OK,                                  Check what's in the index subfiles
OK, creatk
[CREATK rev 17.6]

MINIMUM OPTIONS? yes

FILE NAME? customer
NEW FILE? no

FUNCTION? u

INDEX? Ø

ENTRIES INDEXED:       7
ENTRIES INSERTED:      Ø
ENTRIES DELETED:       2
TOTAL ENTRIES IN FILE:       5

LAST MODIFIED BY MIDAS REV. 17.6

FUNCTION? u

INDEX? 1

ENTRIES INDEXED:       Ø
ENTRIES INSERTED:      Ø
ENTRIES DELETED:       Ø
TOTAL ENTRIES IN FILE:       Ø

LAST MODIFIED BY MIDAS REV. 17.6

FUNCTION? u

INDEX? 2
```

```
ENTRIES INDEXED:       Ø
ENTRIES INSERTED:      Ø
ENTRIES DELETED:       Ø
TOTAL ENTRIES IN FILE:       Ø

LAST MODIFIED BY MIDAS REV. 17.6

FUNCTION? q
OK,                              To get rid off all index subfile
                                 and data subfile entries, zero
OK,                              everything with the ALL option:
OK, kiddel
[KIDDEL rev 17.6]

FILE NAME? customer

DELETE INDEXES:none

ZERO INDEXES: all
OK, creatk
[CREATK rev 17.6]

MINIMUM OPTIONS? yes

FILE NAME? customer
NEW FILE? no

FUNCTION? u

INDEX? Ø

ENTRIES INDEXED:       Ø
ENTRIES INSERTED:      Ø
ENTRIES DELETED:       Ø
TOTAL ENTRIES IN FILE:       Ø

LAST MODIFIED BY MIDAS REV. 17.6

FUNCTION? u

INDEX? 1

ENTRIES INDEXED:       Ø
ENTRIES INSERTED:      Ø
ENTRIES DELETED:       Ø
TOTAL ENTRIES IN FILE:       Ø

LAST MODIFIED BY MIDAS REV. 17.6

FUNCTION? q
OK,                              Everything is now gone.
```

```
OK,

OK, kiddel
[KIDDEL rev 17.6]

FILE NAME? customer

DELETE INDEXES: 2
OK, creatk
[CREATK rev 17.6]

MINIMUM OPTIONS? yes

FILE NAME? customer
NEW FILE? no

FUNCTION? p

INDEX NO.? 2
INDEX DOES NOT EXIST

INDEX NO.? (CR)

FUNCTION? q
OK,
```

To delete an index subfile, use the delete option:

KIDDEL Error Messages

If the MIDAS file being KIDDELed was created by a version of MIDAS earlier than Rev 15, or if other file incompatibilities exist, the KIDDEL utility may return one of several error messages to the user's terminal during file processing.

Some error messages are shared by several MIDAS utilities, so the user may also encounter them while using utilities other than KIDDEL. See Appendix A for a list of these error messages.

Part III
Midas File Access

SECTION 5

FILE ACCESS OVERVIEW

INTRODUCTION

This section is an overview of all the MIDAS file operations that can be performed in the various language interfaces. It summarizes which file access operations are available in each language interface and tells you where to find more information. The next four sections explain how to add data to and retrieve data from a MIDAS file using each of the language interfaces. The file created in Section 2 is used throughout subsequent sections to illustrate MIDAS file access.

Once a simple template is set up, data must be added to create a data subfile. If you already have a sequential data file, Section 3 tells you how to go about "adding" it to the template. There are several ways to do this, the most common being KBUILD. The KBUILD utility is completely documented in Section 3.

As soon as the data subfile has been populated, (that is, when it contains record entries), information can be retrieved from the file using any of the keys defined for the file. Information retrieval, update, deletion etc., are all treated in each of the language interface sections.

Fundamental Access Operations

Accessing a MIDAS file usually involves these basic operations:

- Opening a file for update and/or read access
- Adding a record
- Positioning to a file record on any key
- Positioning to a file record without key; record is automatically locked but not read
- Reading a record by any key (partial/full)
- Reading next record in sequence

- Reading next (sequential) record with the same key
- Locking a record with a read operation
- Updating current record
- Deleting current record
- Deleting a record by a key
- Closing the file

MIDAS Access in Five Languages

Each language interface has a slightly different method of performing these operations. The FORTRAN interface is presented first, as it is the basis of all other MIDAS interface methods and presents the full range of MIDAS access capabilities. In addition, a section of this book is devoted to each of the language interfaces to make life easier for each language programmer:

Section	Language
Section 6	FORTRAN (PMA,F77)
Section 7	COBOL (Keyed index access only)
Section 8	BASIC/VM
Section 9	PL/I Subset G
Section 10	RPGII
Section 12	COBOL (direct access only) (RELATIVE)

Language Interface Routines and Statements

Table 5-1 summarizes the possible MIDAS access functions and the routines and/or statements available to perform them in each language interface. Where an appropriate statement or routine does not exist to perform such an operation, the word "NO" appears. This table is just a summary and does not represent the whole scope of each language interface. Please refer to the indicated section for particulars on the language interface you're interested in.

Table 5-1. Language Interface Routines

Function	COBOL	BASIC	RPG	FORTRAN	PL/I
POSITION FILE and automatically LOCK RECORD: (without returning record)					
by primary	START	{POSITION / REWIND}	CALC SETLL	NO	LOCATE with KEY option
by secondary	START	{POSITION / REWIND}	NO	NO	NO
READ and LOCK RECORD:					
by primary key	READ *	READ {PRIMKEY / KEY}	CALC CHAIN	LOCK$	READ with KEY option
by secondary key	READ *	READ KEY	NO	LOCK$	NO
by partial key	READ *	READ {PRIMKEY / KEY}	NO	LOCK$	NO
next record	READ NEXT *	READ SEQ	Input Cycle or READ +	NEXT$ followed by LOCK$	READ
next with same key value	READ NEXT *	READ SAMEKEY	NO	NEXT$ followed by LOCK$	NO
current record	READ **	READ	NO	LOCK$	NO
READ without LOCK:	READ ***	NO	NO	NEXT$ or FIND$	NO

* Record only locked file is open for I/O
** Only after START or with KEY IS clause (RANDOM access)
*** If file open for ~~OUTPUT~~ INPUT (reading) only
\+ For indexed files declared as Primary (P), Secondary (S), or Demand (D)

Table 5-1. Language Interface Routines (continued)

Function	COBOL	BASIC	RPG	FORTRAN	PL/I
READ KEY: (read key associated with record)					
current key	NO	READ KEY	NO	FIND$	READ with KEYTO(var)
keyed on partial primary or secondary	NO	READ KEY	NO	FIND$	NO
READ SECONDARY DATA:	NO	NO	NO	FIND$	NO
UPDATE/REWRITE current data record	REWRITE	UPDATE	Output Cycle	UPDAT$	REWRITE [key optional]
ADD:					
data record and keys by primary key	WRITE	ADD	Output: ADD	ADD1$	WRITE
secondary index to a data record	NO	NO	NO	ADD1$	NO
DELETE:					
current data record	DELETE *	NO	NO	DELET$	DELETE [no key]
data record by primary key	DELETE	REMOVE	NO	DELET$	DELETE with KEY option
secondary index entry (only)	NO	REMOVE	NO	DELET$	NO
UNLOCK record without update:	NO	NO	NO	UPDAT$	NO

* In SEQUENTIAL access, only if record already locked by a READ

General Changes

The file-no argument, which appears in all of the FORTRAN subroutine calling sequences, can be set to any value the user desires. Currently the documentation infers that file-no should be set to 0. This argument is ignored completely by MIDAS and is being preserved only for compatibility.

SECTION 6

THE FORTRAN INTERFACE

INTRODUCTION

The FORTRAN interface to MIDAS is a collection of user-callable routines that can be called from any program written in FORTRAN, PMA, PL/I, and so forth. However, it only makes sense to use these routines in FORTRAN and PMA, as all the other programming languages that support MIDAS have already incorporated these routines into their respective interfaces. The FORTRAN/MIDAS interface package handles both keyed-index and direct access MIDAS files. The main emphasis of this section is on keyed-index access; therefore, this information is presented first, and is followed by pertinent direct access information. Background information on direct access MIDAS files, including a description of file structure and how to create a direct access template, is covered in Section 11.

Which FORTRAN?

The term "FORTRAN", as used in this section, refers to both FORTRAN IV and FORTRAN 77, the two Prime-supported versions of the FORTRAN programming language. Both versions of FORTRAN handle MIDAS calls identically, and programs can be written in either language to access MIDAS files. One should be careful of the differences between the two languages, however. There is no guarantee that all programs written and compiled under FTN will compile and run under F77. For example, one of the commonly-encountered discrepancies between the two involves variables declared as "INTEGER". FORTAN IV assumes they are INTEGER*2 and FORTRAN 77 assumes they are INTEGER*4. Refer to The FORTRAN 77 Reference Guide for a summary of the important differences between the two languages. All the programs shown in this section were compiled without errors under both compilers.

Compile and Load Sequence

All FORTRAN programs that use MIDAS must have the MIDAS library VKDALB, included in the SEG load sequence, as shown in this example. User input is underlined to distinguish it from system output.

```
OK, FTN PROGRAM -64V
0000 ERRORS [<.MAIN.>FTN-REV17.6]

OK, SEG
[SEG rev 17.6]
# LOAD #PROGRAM
$ LO B_PROGRAM
$ LI VKDALB
$ LI
LOAD COMPLETE
$ SA
$ Q

OK,
```

The same sequence applies to programs written in FORTRAN 77; simply substitute "F77" for "FTN". Additional FORTRAN program requirements are addressed under $INSERT Mnemonics later in this section.

FORTRAN's Requirements

All the other language interfaces sit on top of the FORTRAN interface, automatically taking care of all the subroutine-level file I/O, eliminating the need for the programmer to keep track of what's actually going on "underneath." The FORTRAN interface, however, requires the user to be concerned with explicit tasks like notifying MIDAS when a file is opened and closed, permitting several users to access the same file simultaneously or denying multiple user access to that file, locking/unlocking a record, keeping track of the current record (see FILE ACCESS CONCEPTS, below) and monitoring file position relative to an index subfile.

Why the Tasks are Necessary

The special method of opening and closing files is part of the MIDAS concurrent process handler which regulates the simultaneous access of processes to a single MIDAS file. There are several methods of doing this, all of which are described in this section. The current record must be monitored by MIDAS in order to avoid errors like operating on the wrong record. To keep track of current file position and of what the user is doing in a given file, MIDAS uses a 14-word array. While the entire array is not the user's responsibility, the user must know how to utilize certain parts of it. These tasks aren't at all difficult, although they may seem complicated at first. They are explained a bit at a time so you can see how they all fit together.

Summary of FORTRAN Access Operations

Below is a summary of the basic access operations available to the FORTRAN (and PMA) programmer. The subroutines that perform these operations are listed below, and are described fully later in this section.

FORTRAN Access Operations and Corresponding Subroutines

Operation	FTN Subroutine
Adding a record	ADD1$
Adding a secondary index entry	ADD1$
Closing a file	CLOSM$,NTFYM$
Deleting a record	DELET$
Deleting an index entry	DELET$
Locking a record for updating	LOCK$
Opening the file	OPENM$,NTFYM$
Positioning the file by key	FIND$,NEXT$
Reading a record by key	FIND$,NEXT$
Reading duplicates	NEXT$
Reading next record	NEXT$,GDATA$
Updating a record	UPDAT$

OPENING AND CLOSING MIDAS FILES

Like all FORTRAN files, MIDAS files must be explicitly opened and closed. There are two ways to open a file in the current version of MIDAS (this applies to versions stamped Rev. 17 and above):

- Existing programs which use SRCH$$ or TSRC$$ to open and close the file can be modified by making calls to NTFYM$, a routine that notifies MIDAS when a MIDAS file is opened and/or closed.

- The OPENM$ subroutine can be used to perform the tasks of both SRCH$$ (or TSRC$$) and NTFYM$ in one step.

In either case, opening the file associates it with a particular file unit, which is in turn associated with the user number of the process which opened that file unit.

Setting the READ/WRITE Lock

The READ/WRITE lock on all MIDAS files should be set to 3 before they are accessed. See your System Administrator if there are any questions. This READ/WRITE lock setting allows a MIDAS file to be opened for writing and reading by more than one process at a time. This is part of MIDAS's concurrent process handler, which is explained in Section 13 and Appendix D.

Requirements for Existing Applications

Existing FORTRAN application programs (written prior to Rev. 17) must be modified in order to operate with the current version of MIDAS. MIDAS must be notified when a MIDAS file (segment directory) has been opened for processing. You can replace the existing file open/close routine (generally it's SRCH$$ or TSRC$$) with calls to OPENM$ and CLOSM$; alternatively, leave the existing open/close routine in place and insert calls to NTFYM$ after the file has been opened and before the file is to be closed. If desired, the user can completely disable concurrent process handling, thereby making no changes to existing application programs. This, however, will result in performance degradation. See Section 13 for details on disabling the present method of concurrency handling.

Benefits of "New" Method: OPENM$, CLOSM$ and NTFYM$ allow segments to be left open between calls, resulting in a great performance improvement over the old method. (See Previous Concurrency-Handling Methods, in Appendix D.) They also permit MIDAS's concurrent process mechanism to work, which allows more than one user to have the same segment subfile open for use. However, it prevents these processes from operating on the same record simultaneously. If you do not use either OPENM$ and CLOSM$, or NTFYM$ and SRCH$$, the new concurrent process handling method will be disabled. The old methods of opening and closing files do not allow for file segments being left open between calls which results in significant peformance degradation.

Note

Programs that use offline routines (documented in Section 14) should not use OPENM$/CLOSM$ or NTFYM$ to open MIDAS files. Use SRCH$$ or TSRC$$ instead.

OPENM$: OPENING THE FILE

OPENM$ is the MIDAS routine which opens a MIDAS file (segment directory), associates it with a file unit, and notifies MIDAS that processing is about to begin on that particular file. MIDAS itself needs this information to keep track of all the processes interacting with a MIDAS file; without this "monitoring" system, MIDAS file integrity could be adversely affected by conflicting concurrent processes.

OPENM$ Keys

OPENM$ replaces direct calls to either of these PRIMOS file system routines:

- SRCH$$, which takes a filename argument
- TSRC$$, which takes a pathname or treename argument

These routines are responsible for opening a file and associating it with a PRIMOS file unit. OPENM$ requires the use of certain TSRC$$ or SRCH$$ keys, which tell PRIMOS whether a file is to be opened for reading, writing or both:

Key	Action
K$GETU	Opens file on an available PRIMOS file unit; should be used when first opening the file from a program
K$READ	Opens file for reading only
K$WRIT	Opens file for writing only
K$RDWR	Opens file for reading and writing

These keys are used in calling OPENM$ as shown below.

OPENM$ Calling Sequence

The calling sequence of OPENM$ is:

```
CALL OPENM$ (key, pathname, namlen, funit, status)
```

The arguments, which are all INTEGER*2, are:

key — Valid OPENM$ access key: K$READ, K$WRIT, or K$RDWR, used optionally together with K$GETU (supplied by user)

pathname — Pathname (treename) of MIDAS file to be opened (supplied by user)

namlen — Length of pathname in characters (supplied by user)

funit — funit is the file unit on which the file is to be opened (supplied by user); if K$GETU is specified instead, funit is returned as the file unit on which the file was opened (returned by OPENM$)

status — Error status (returned by OPENM$)

Ø	No error
< 10001	PRIMOS file system error (system-defined)
10001	Bad key
10002	Too many MIDAS files open: limit is 128 (This is the MFILES argument in KPARAM file: see Section 15.)
10003	Specified file is not a MIDAS segment directory

CLOSM$: CLOSING THE FILE

The CLOSM$ routine closes a MIDAS file (segment directory) opened on a specified file unit, and closes any of the subfiles which MIDAS has opened during file access.

CLOSM$ Calling Sequence

The calling sequence of CLOSM$ is:

```
CALL CLOSM$ (funit, status)
```

The arguments (INTEGER*2) used in this call are:

funit — File unit on which the MIDAS file is open (supplied by user)

status — Error status (returned by CLOSM$)

- Ø — No error
- > Ø — PRIMOS file system error (system-defined)

Using OPENM$ and CLOSM$

The following program segment could be used to open and close the MIDAS file created in Section 2:

```
C     Use OPENM$ to tell MIDAS we're
C     going to open and use this file
      CALL OPENM$ (K$RDWR+K$GETU,'CUSTOMER',8,FUNIT,STATUS)
      .
      .
      .
C     Process MIDAS file (with FIND$,ADD1$ etc.)
      .
      .
C     Close the file
      CALL CLOSM$(UNIT,STATUS)
      .
      .
      END
```

In the above example, the K$GETU key tells PRIMOS to open the file CUSTOMER on any available file unit that is returned in FUNIT. STATUS returns an error code that tells what kind of error, if any, was encountered upon opening or closing the file.

NTFYM$: THE "NOTIFY MIDAS" ROUTINE

The NTFYM$ routine informs MIDAS that a MIDAS file (segment directory) has been opened or is about to be closed by the user. It can generally be inserted into an existing program immediately after a call to SRCH$$ is made to open the file and immediately before SRCH$$ is again called to close the file.

A call to NTFYM$ after a MIDAS file has been opened tells MIDAS that it should leave open between MIDAS calls any of the file's segment subfiles opened during subsequent file access. A call to NTFYM$ before a MIDAS file is closed tells MIDAS that it should close any of the file's segment subfiles that it has left open. If the MIDAS library has been customized to disable internal locking, a call to NTFYM$ has no effect. Users with existing applications may use NTFYM$ along with SRCH$$ or TSRC$$ (in lieu of OPENM$ and CLOSM$).

NTFYM$ Calling Sequence

The calling sequence is:

```
CALL NTFYM$ (key, unit, status)
```

The arguments (all INTEGER*2) used in this call are:

key — specifies whether the file has been opened or is about to be closed (supplied by user)

- 1 - file has been opened
- 2 - file is about to be closed

unit — File unit on which the file is open (supplied by user)

status — Error status (returned by NTFYM$)

- Ø — No error
- 10001 — Bad argument
- 10002 — Too many MIDAS files open simultaneously; occurs only if key is 1 (default limit = 128)

Use of NTFYM$

The following sample FORTRAN program opens the CUSTOMER file on a file unit obtained by using the K$GETU key. (See OPENM$ Keys, above.) Assuming the call to NTFYM$ is successful, the variable TYPE returns a code identifying the type of the file that was opened. If the file is indeed a MIDAS file, TYPE will be returned with a value of 2, indicating that the file is a SAM segment directory. (See The PRIMOS Subroutines Reference Guide for more information.) The program calls NTFYM$ to notify MIDAS that it is ready to begin operations on a MIDAS file. The program also notifies MIDAS when processing is completed and then closes the file.

```
C     OPEN THE FILE
      CALL SRCH$$ (K$READ,'CUSTOMER',8,K$GETU,TYPE,CODE)
      IF (CODE .NE. Ø) GO TO 9ØØØ          /* ERROR ON CALL
      IF (TYPE .NE. 2) GO TO 9999          /* NOT A MIDAS FILE
      CALL NTFYM$(1,UNIT,CODE)             /* TELL MIDAS WE'RE READY
      IF (CODE .NE. Ø) GO TO 9ØØ2          /* HANDLE ERRORS
2ØØ   CONTINUE
      .
      .
      .
C     DO MIDAS FILE PROCESSING (E.G. CALLS TO FIND$ ETC.)
      .
      .
      .
      CALL NTFYM$ (2,UNIT,CODE)            /* TELL MIDAS WE'RE DONE
      CALL SRCH$$ (K$CLOS,Ø,Ø,UNIT,TYPE,CODE) /* CLOSE FILE
      .
      .
      .
     ETC.
```

The labels 9ØØØ, 9ØØ2, and 9999 refer to statements in the program (not shown here) that handle errors occurring on calls to SRCH$$ and NTFYM$. Note that if we'd used a pathname for the MIDAS file instead of a filename, TSRC$$ would have been used in place of SRCH$$.

FILE ACCESS CONCEPTS

MIDAS file access at the FORTRAN call-interface level involves some important concepts which should be understood if you want to write and debug a program successfully. These concepts include:

- The current record
- Record locking
- The communications array
- $INSERT files
- The MIDAS flags

The Current Record

The current record can be thought of as that record in the file to which the file pointer is presently positioned. Usually, this occurs as a result of a read (find) operation. Some MIDAS calls need to know which record is the current record so that the operation is performed on the right one. For example, if a record is being read, it is the current record. After the read operation is complete, that "current record" location is stored away so the next operation knows which record to act on should that be necessary. If the subsequent operation is a "read next" operation, the file handler has to check where the current file position is so it can read the proper record. The proper record is the one after the record just read, so it becomes the current record. If, however, the subsequent operation is a call to FIND$, MIDAS doesn't care which record is current because it must do an index search to find the desired record anyway.

This current record position information is stored in 14-word array supplied on each MIDAS call. It is constantly updated and checked by MIDAS, and is used to communicate index location, file position and current record location between MIDAS and the user. Consequently, it is called the MIDAS "communications array," although it is commonly referred to simply as "the array" throughout this book.

The Communications Array

The communications array is used by MIDAS to keep track of the current file position once a MIDAS file has been opened for access. The array stores several items, including the address of the current record, the current position in the index subfile, an indication of whether or not the last record was found, the word number of the located entry in the index subfile, and the data record address. The size of the array is 14 words, but only the first five are important in the basic operations

dealt with here. The array is one of the arguments used in the calling sequences of most FORTRAN/MIDAS file access subroutines, so you'll be seeing a lot of it throughout this section.

Record Locking

In order to update a record, it is necessary to lock that record for exclusive use. This prevents anyone else from attempting to read or change the record while you are changing it. The other language interfaces to MIDAS perform locking automatically; the method of doing so varies in each one. Only FORTRAN requires that an explicit lock action be taken by calling a special subroutine, LOCK$, before an update operation can occur. Record locking, however, does not entirely eliminate the possibility of concurrency errors because a locked record can be deleted by another user.

The Array Format: From the user's standpoint, only the first five words of the array are really important. Table 6-1 describes the first five words of the array as implemented for keyed-index access (access to keyed-index MIDAS files). The description includes what each word indicates and the meaning of the important bits in each word. The format of the array as used in direct access is shown in Table 6-2. The complete format of the array is discussed in Section 13.

Only the first word of the keyed-index access array can be modified by the user; the others are taken care of by MIDAS and return information that may or may not be important to your application.

Table 6-1. Keyed-Index Access Array Format

Word Number	Describes
Word 1	When supplied by user, can be Ø, 1, or -1 (see below); when returned by MIDAS, contains array state code
Words 2-4	Current position in index subfile (index entry address)
Word 2, bits 1-8	Entry number
Word 2, bits 9-16	Segment file number
Words 3-4 (32 bits)	Word offset of index subfile block
Word 5	Hash value (based on current key value)

Word 1: Input Value: The only word in the array that can be modified by the user is word 1. It can be set by the user to a value of 1, Ø, or -1. Any other value produces an error on any call in which that array is used. If set to Ø or 1, MIDAS is told to use the current array contents on this call. If set to -1, MIDAS is told to ignore the array contents.

Word 1: Output Value: The first word in the array is always used by MIDAS to return a completion code after an operation has finished. If set to Ø or 1, the array contents are valid, and no error was flagged on the last call. If there was an error on the last call, word 1 has a value greater than 1 corresponding to some MIDAS error condition code. Appendix A contains a complete list of MIDAS error codes.

The Direct Access Array

The direct access array format differs slightly from the one used in keyed-index access. The user must supply values for words 2, 3 and 4 of the array on some calls. It is important to supply the proper values for each word of the array when processing direct access MIDAS files.

Table 6-2. Direct Access Array Format

Word No.	Setting	Meaning
1	Ø or 1	Use array contents (supplied by user)
2	entry size (in words)	Entry length is the primary key length in words, plus data record length in words, plus 2 words (supplied by user)
3-4	record number	A single-precision (REAL*4) floating-point record number (supplied by user)
5-14		Same as for keyed-index access

$INSERT Mnemonics

Another of the concepts peculiar to the FORTRAN interface is the $INSERT file SYSCOM>PARM.K. It contains a host of parameters used by MIDAS in the interface FORTRAN subroutines and must be inserted in each FORTRAN program that uses MIDAS. Put the statement:

```
$INSERT SYSCOM>PARM.K
```

at the beginning of your programs. The $INSERT statement must begin in column 1. Most of the parameters in the PARM.K file which you are

likely to encounter are described later under The MIDAS Flags.

Another file the user will need to insert at the beginning of a FORTRAN program is the SYSCOM>KEYS.F file which contains declarations for all the keys used by the FORTRAN-MIDAS interface subroutines. Use the statement:

```
$INSERT SYSCOM>KEYS.F
```

at the beginning of all your FORTRAN programs. PMA programmers should use the file SYSCOM>KEYS.P instead.

The MIDAS Flags

MIDAS has a special set of flag values which can be set in each of the various subroutine calls to tell MIDAS what options are to be used for that call. Options are specified with a set of flag names which are defined in the insert file SYSCOM>PARM.K. The flag names correspond to single bits of a one-word parameter called flags, which is passed to MIDAS by the user in each subroutine call. Table 6-3 lists each flag name and the bit to which it corresponds. To use the flag options, compile your programs with the statement:

```
$INSERT SYSCOM>PARM.K
```

Each flag bit is set off by default; you set certain ones on before a call, depending on what you want to do on that particular call. To set a flag on, all you have to do is specify the name of that flag in an assignment statement or in place of the flags argument on the actual call. MIDAS takes care of the rest. For example:

```
FLAGS = FL$FST + FL$RET
```

As a result of this assignment, the octal values of the two flags FL$FST and FL$RET are added together, and the result, a single octal value, determines which bits are set off and which are set on in the flag word. Don't worry about the octal values because all the bits are initialized for you in the PARM.K file. The bit settings indicate the actions to be taken on the call.

All the flag names, the bits to which they correspond, their octal values, their meanings when set off or on, and the subroutine calls in which they can be used, are listed in Table 6-3.

Table 6-3. MIDAS Flag Names, Values and Meanings

Bit No.	Name	Setting	Meaning	Can Be Used In
1.	FL$USE	ON	Use current copy of array.	All calls
	(:100000)	OFF	Don't use current copy of array.	
2.	FL$RET	ON	Return entire array for use on subsequent calls.	All calls
	(:40000)	OFF	Return completion code only, in array(1).	
3.	FL$KEY (:20000)	ON	On calls to FIND$, NEXT$ and LOCK$, returns primary key with data record. On calls to ADD1$, tells MIDAS not to make its own copy of the primary key to store in data record. Used only if primary key is first field in data record.	All calls
		OFF	On calls to FIND$, NEXT$ and LOCK$, does not return primary key in data buffer. In ADD1$, tells MIDAS to store a copy of primary key in each data subfile record.	
4.	FL$BIT	ON	Call specifies key size in bits, if key is a bit string, or in bytes, if it is an ASCII key.	FIND$ and NEXT$
	(:10000)	OFF	Call specifies key size in words (default).	
5.	FL$PLW (:4000)	ON	Position to next index entry greater than or equal to current or user-supplied entry.	FIND$ and NEXT$
		OFF	Position to next index entry only if it matches current or user-supplied key.	

6.	FL$UKY (:2000)	ON	Update user-supplied key field with version stored in the file. Useful in partial key searches.	FIND and NEXT$
		OFF	Don't update user-supplied key field.	
7.	FL$SEC (:1000)	ON	Return secondary data instead of data record.	FIND$ and NEXT$
		OFF	Return data record read from data subfile.	
8.	FL$ULK (:400)	ON	Unlock data entry only -- don't update it.	UPDAT$ only
		OFF	Update data entry and unlock it.	
9.	FL$FST (:200)	ON	Position to first index entry in subfile.	FIND$, LOCK$, and NEXT$
		OFF	Position to first entry that matches current entry or user-supplied key value.	
10.	FL$NXT (:100)	ON	Position to next index entry greater than current entry or user-supplied key value.	FIND$ and NEXT$
		OFF	Position to next index entry that matches current entry or user-supplied key value.	
11.	FL$PRE (:40)	ON	Position to index entry just prior to current index entry or user-supplied key value.	FIND$ and NEXT$
		OFF	Position to index entry that matches the current or user-supplied key value.	

Note

Bits 12-16 of the flags parameter must be set to 0, the default setting, at all times. Leave them alone.

Order of Precedence: When specified in combination, certain flags have precedence over others. The priority order is:

- FL$FST
- FL$NXT
- FL$PLW
- FL$PRE

Observe that certain combinations of flags are simply not sensible: for example, FL$NXT and FL$PRE. Although meanings are given for each flag when set off, the actual action taken on any given call is dictated by the combination of keys set on, which may, in fact, override the defaults.

THE FORTRAN/MIDAS INTERFACE SUBROUTINES

There are seven FORTRAN subroutines that can be called directly by the FORTRAN programmer in accessing a MIDAS file; they are used (transparently to the user) by all the other language interfaces. Most of these subroutines share the same calling sequence, which makes it easier to learn how to use them. They're listed below by function:

Function	Subroutine
Adds a data record	ADD1$
Adds a secondary index entry	ADD1$
Deletes a data entry or secondary index entry	DELET$
Finds a data entry by any key	FIND$
Finds the next data entry via an index	NEXT
Locks a data entry for update	LOCK$
Reads data entries in order stored (physically)	GDATA$
Updates a data entry	UPDAT$

User-Supplied Information

The MIDAS access subroutines (ADD1$, FIND$, LOCK$, DELET$, NEXT$ and UPDAT$) use essentially the same calling sequence. The variables used in this calling sequence pass the following information along to MIDAS:

- The file unit number on which the MIDAS file is open
- The size of the data record buffer: used for data added to or returned from the data subfile
- The size of the primary key buffer
- The MIDAS communications array
- The flag argument specifying the options to be used in the call (see below)
- A program label indicating the alternate return to be taken in the event of an error: set to Ø if no alternate return exists
- The access method to be used (keyed-index or direct access)
- The type of index subfile to be used (primary or secondary)
- The length of the data to be transferred (except in DELET$ calls)
- The length of the key to be used in partial key access

General Calling Sequence

The general format of the calling sequence for the six data access routines mentioned above is:

CALL routine (funit, buffer, key, array, flags, altrtn, index, file-no, bufsiz, keysiz)

Argument	Data Type	Specifies
routine	character	One of these six routines: ADD1$, DELET$, FIND$, LOCK$, NEXT$ or UPDAT$
funit	integer (short)	The file unit on which the MIDAS file is open
buffer	integer (short)	The [illegible] data record buffer into which data is read or from which it is written to the file
key	integer (short)	The key value to be used on call
array	integer (short)	The communications array which holds current record and index position information: also returns status codes after each call
flags	integer (short)	The flag word specifying options for this call: see below
altrtn	integer (short)	Label in program of alternate return to be taken if an error occurs (set to Ø if no alternate return exists)
index	integer (short)	The access method to be used, (keyed-index or direct access) and which index to use if not direct access
file-no	integer (short)	Obsolete: set to Ø
bufsiz	integer (short)	The length of the data to be transferred to/from file (except in calls to DELET$): set at Ø if full data entry is being transferred
keysiz	integer (short)	The length of the key to be used in partial key access (used with FIND$ and NEXT$ only)

SUPPLIED IN WORDS.

"Optional" Arguments: The arguments for which the user may supply a Ø instead of another value, and their respective meanings when set to Ø are:

Argument	Default
altrtn	There is no alternate return for handling errors on this call
file-no	Obsolete -- maintained for compatibility
bufsiz	Default data subfile entry length (stored in file)
keysiz	Defaults to key length specified in file: can be set at Ø if full key is being used

Note

Another data access subroutine, GDATA$, which is used for sequential retrieval of entries in the data subfile, does not use the general calling sequence just described. Its particular calling sequence and function are described later in this section.

ADD1$

The ADD1$ routine is used to add primary index entries and data subfile entries to keyed-index and direct access MIDAS files. It also adds secondary index entries (and optional secondary data) to files that have secondary indexes. During an add (as in all file operations), the file is effectively locked to other users to prevent unnecessary complications.

Keyed-Index Adds

Records can only be added to a MIDAS file when accompanied by a primary key value. Similarly, a secondary key value can only be added if the record which it will reference already exists in the data subfile and is referenced by a primary index entry. Secondary index entries are always added separately from the primary index and data entries. Generally, records and the keys associated with them are added in this sequence:

1. First, make a call to ADD1$, with FL$RET in flags set on, to add a data entry and its primary key value.

2. Make a separate call to ADD1$, with FL$USE set in flags, to add a secondary index entry for this record.

You can also add secondary index entries for an existing data subfile record at a later time, in either of two ways. The first method is to supply the primary key value in buffer and set index and key to the desired secondary index number and value respectively on a call to ADD1$. In the second method, the record must first be located by primary key (with a FIND$ or NEXT$ call). The array must be returned by this call so that the call to ADD1$ will be able to use it. index and key must be set to the index subfile number and value on the call to ADD1$.

ADD1$ Calling Sequence

The calling sequence uses the general format shown earlier:

```
CALL ADD1$ (funit, buffer, key, array, flags, altrtn, index,
            file-no, bufsiz, keysiz)
```

Some of the arguments have meanings peculiar to ADD1$. They are explained in Table 6-4.

Index Values: The index argument indicates whether the add is being performed on a direct access file or a keyed-index file. It also tells whether a primary or secondary index entry is to be processed on this call. Here is a summary of the values for the index argument:

Value for Index	Meaning
Ø	Primary index
1 - 17	Secondary index
-1	Direct access

When index is Ø, the data entry information to be added to the data subfile must be supplied in buffer. If you are storing keys in the data, be sure to incude all-key values in buffer. When adding secondary indexes, index should be a number from 1 to 17, the corresponding primary key must be specified as the first item in the buffer; the secondary key value must be specified in full in the key argument.

On pages 6-21 and 6-31, the argument file-no was mistakenly omitted from the argument explanation list. Please insert the following in between the index and bufsiz arguments on these pages:

file-no Set this to 0: obsolete.

Table 6-4. ADD$1 Arguments

Argument	Meaning
funit	The file unit on which the MIDAS file is opened.
buffer	On a data entry add, it contains the data subfile record. If keys are being stored in the data record, include all key values in buffer as well. On a secondary index add without FL$USE set, buffer contains the primary key value followed by optional secondary data.
key	Must contain full primary key value on a data record add; contains full secondary key value on a secondary index add.
array	The communications array.
flags	The flags options for calls to ADD1$ are shown in Table 6-5.
altrtn	Statement number of alternate return to be taken in event of error; supply Ø if no alternate return.
index	Indicates access method and index subfile to use: Ø = primary index 1-17 = secondary index -1 = direct access
bufsiz	Length of data to be added to subfile from buffer. If bufsiz = Ø and index = Ø, MIDAS adds data from buffer to data subfile, taking only the number of words it needs to match the record size defined for the MIDAS file during CREATK. If index > Ø, adds secondary data from buffer to indicated index subfile, if secondary data is supported for that index. If bufsiz is less than data record size or is less than key size plus secondary data, only that part of buffer will be used. The rest of data subfile or secondary data entry is zero-filled. In general, specify Ø for fixed-length records: for files with variable-length records, specify length of data to be written to file.
keysiz	Set to Ø (ignored).

Table 6-5. Flags for ADD1$

Flag	Meaning
FL$USE:	When set on, uses contents of array from previous call -- used on calls to add secondary index entries. Set off when adding primary index entries or secondary index entries when FL$USE is off.
FL$RET:	When set on, returns array contents from this call (set only on calls to add data records).
FL$KEY:	When set on, tells MIDAS not to store its own copy of the primary key with each data subfile record; see Redundant Primary Keys, below.

Redundant Primary Keys

MIDAS always stores its own copy of the primary key with each entry in the data subfile. When an entry is returned from the data subfile, this primary key value is not returned unless specifically requested by the user. However, if keys are being stored in the data record (by the user), it is not necessary to have an extra copy of the primary key, as long as the user's copy of the primary key appears in the beginning of the data record. To eliminate redundant copies of the primary key, set FL$KEY on in FLAGS for all ADD1$ calls that add data subfile entries. Remember, this option should be used only if the primary key is the first field in the data subfile record.

Adding Data Records

When adding primary index and data entries, the full primary key value associated with the record to be added should be placed in key. The information to be added to the data subfile is placed in buffer. The length of this entry must be supplied in bufsiz. For keyed-index MIDAS files with fixed-length records, bufsiz should be set to Ø; for variable-length records, set bufsiz to the length of the data entry (in words). If there are secondary indexes in this file, the array should be returned (set the FL$RET flag on in flags) for use in subsequent calls to ADD1$.

Note

When adding entries to a MIDAS file with ADD1$, the full key value must always be supplied in the key argument. Partial key values are illegal. The argument keysiz is therefore ignored and can be specified as Ø if desired. Consequently, bit 4 (FL$BIT) of flags is not relevant in calls to ADD1$.

Adding Secondary Index Entries

When adding secondary index values, the user must provide MIDAS with this information:

- Secondary index number (in index)
- Secondary key value (in key)
- Primary key value -- must either be in first part of buffer or FL$USE must be set to use a valid copy of array. The array is valid only if returned by an immediately prior call in which the desired key value was used and/or returned.
- Secondary data (optional -- supplied in buffer, following primary key value)

The full primary key value is always supplied as the first field in buffer, if FL$USE is not indicated. Any secondary data information should be included after the primary key in the buffer argument.

Duplicate secondary key entries are supported only for those index subfiles which were created with duplicate status during CREATK. An attempt to add duplicate entries to a secondary index that does not support them results in an error and the failure of the add operation.

Typical Scenario: If you want to add all secondary index entries for a particular data subfile entry immediately after you've added the primary key and the data entry, here's the typical sequence of events you'd follow:

1. Set FL$RET in flags on the ADD1$ call when adding the primary index and data entry.

2. After the above call, set FL$USE in flags if you have one or more secondary index entries to add.

3. Set index to the appropriate index subfile number (1 -17).

4. If a valid array exists from a previous call (see Adding Secondary Index Entries above), simply set FL$USE. Or, if the array is not valid, put the primary key value of this record in the first part of buffer. This won't be necessary if you store keys (in order) in the data record because you will already have put the primary key value in buffer(1) on the previous call.

5. Put any secondary data for this index entry in buffer, immediately following the primary key value.

6. Set key to the full secondary key value you want stored in the index subfile.

7. Make the call to ADD1$ to add this secondary index entry.

8. Repeat steps 2 - 7 for each secondary index entry to be added for this record.

ADD1$ Example

The following program adds entries to the MIDAS file CUSTOMER (set up in Section 2). It is an interactive program intended for on-line data entry. The program first asks the user for a primary key value, then asks if there are any secondary key values to be added for this record. After key values have been accepted, it asks for the non-key portion of the data record to be stored in the index subfile. In this application, all keys are being stored in the data record, and the primary key is the first field in the data record. If desired, we could use the FL$KEY option on each add so that MIDAS will use the data record's copy of the primary key instead of making its own copy of this key. However, we would always have to set FL$KEY in calls to FIND$ or NEXT$ in order to get the primary key returned with the data record if we chose this option. Instead, FL$KEY is set off in this program so the full data record is always returned.

```
C     ADD PROGRAM FOR CUSTOMER FILE
C
C     THIS PROGRAM ADDS ENTRIES TO THE CUSTOMER FILE BY ASKING THE
C     USER FOR KEY VALUES AND DATA RECORD VALUES.
C     THE KEYS ARE STORED IN THE DATA RECORD.
C
$INSERT SYSCOM>KEYS.F
$INSERT SYSCOM>PARM.K
C     DECLARATIONS
C     Garden variety call parameters
      INTEGER*2 ARRAY(14),INDEX,FUNIT,BUFFER(35),STATUS,FLAGS
      INTEGER*2  KEY(13),BUFSIZ,KEYSIZ,CODE
C     KEY IS MAX OF 13 WORDS (SEC KEY01)
C
      INTEGER*2 ANSWER,                              /* YES OR NO
     *  I,                                           /* LOOP INDEX
     *  PKEY(3),                                     /* PRIMARY KEY
     *  SKEY1(13),                                   /* SEC. KEY01
     *  SKEY2(2),                                    /* SEC. KEY02
     *  DATA(17)                                     /* JUST DATA PART
      LOGICAL *2 SWITCH(2)                           /* TELLS WHEN WE
C     HAVE SECONDARY KEYS TO ADD FOR AN ENTRY
C     SET SWITCH(1)  ON IF WE HAVE ENTRY FOR SEC KEY01
C     SET SWITCH(2) ON IF WE HAVE ENTRY FOR SEC KEY02
C
C     EQUIVALENCE KEYS AND DATA TO PARTS OF BUFFER
C     SO THAT ALL KEY VALUES WILL BE STORED WITH RECORD
      EQUIVALENCE (BUFFER(1),PKEY(1)),
     * (BUFFER(4),SKEY1(1)),
     * (BUFFER(17),SKEY2(1)),
     * (BUFFER(19),DATA(1))
C
      SWITCH(1) = .FALSE.                            /* SET OFF INITIALLY
      SWITCH(2) = .FALSE.
C     OPEN FILE
```

```
10    CALL OPENM$(K$RDWR+K$GETU,'CUSTOMER',8,FUNIT,STATUS)
      IF(STATUS .NE. 0) GO TO 100              /* ERROR
C
20    CALL TNOUA('ENTER PRIMARY KEY VALUE: ',25)
      READ(1,2222)PKEY
C
C     ASK IF THERE ARE SECONDARIES TO ADD
30    CALL TNOUA('ANY SECONDARIES TO ADD FOR THIS RECORD?',40)
      READ(1,1111)ANSWER
      IF(ANSWER .EQ. 'N') GO TO 50
C     ELSE GO ON
C
35    CALL TNOUA('ENTER INDEX NO.: ',17)
      READ(1,6666)INDEX
      CALL TNOUA('ENTER KEY VALUE: ',17)
      IF(INDEX .EQ. 2) GO TO 40
      READ (1,4444)SKEY1
      SWITCH(1) = .TRUE.
      GO TO 45
40    READ(1,5555)SKEY2
      SWITCH(2) = .TRUE.                       /* ENTRY FOR INDEX 02
45    CALL TNOUA('MORE?',5)
      READ(1,1111)ANSWER
      IF(ANSWER .EQ. 'Y') GO TO 35
C     ELSE GO ON
C
50    CALL TNOUA('ENTER DATA RECORD (NON-KEY) INFO: ', 34)
      READ(1,3333)DATA
C     SET UP FLAGS AND OTHER ARGS FOR CALL TO ADD1$
      INDEX = 0
      FLAGS = FL$RET
60    CALL ADD1$(FUNIT,BUFFER,PKEY,ARRAY,FLAGS,$200,INDEX,0,0,0)
C
      IF( .NOT. SWITCH(1)) GO TO 90              /* NO KEY01 ENTRY
C     ELSE ADD ENTRY TO INDEX SUBFILE 01
C
75    FLAGS = FL$USE + FL$RET                    /* USE ARRAY
      CALL ADD1$(FUNIT,BUFFER,SKEY1,ARRAY,FLAGS,$200,1,0,0,0)
C
80    IF( .NOT. SWITCH(2)) GO TO 90          /* ENTRY FOR INDEX 02 ?
C     IF SO, ADD ENTRY FOR INDEX 02, RE-USING FLAGS SETTING
      CALL ADD1$(FUNIT,BUFFER,SKEY2,ARRAY,FLAGS,$200,2,0,0,0)
C     SEE IF WE'RE DONE
90    CALL TNOUA('READY TO QUIT? (Y OR N)',23)
      READ(1,1111)ANSWER
      IF(ANSWER .EQ. 'N') GO TO 20
C     ELSE GET OUT
      GO TO 444                                  /* CLOSE FILE
C     FORMATS
C
1111  FORMAT(A2)                                 /* ANSWER
2222  FORMAT(6A2)                                /* PKEY
3333  FORMAT(34A2)                               /* DATA
4444  FORMAT(26A2)                               /* SKEY1
```

```
5555  FORMAT(4A2)                                  /* SKEY2
6666  FORMAT(I1)                                   /* INDEX
C
C
C     ERROR HANDLERS
100   CALL TNOUA('ERROR ON OPEN: STATUS IS: ',24)
      CALL TODEC(STATUS)
      GO TO 444
200   CALL TNOU('ERROR ON ADD',12)
      CALL TNOUA('INDEX IS: ',10)
      CALL TODEC(INDEX)
      CALL TONL
      CALL TNOUA('ARRAY(1) IS: ',13)
      CALL TODEC(ARRAY(1))
      CALL TONL
      GO TO 20
444   CALL CLOSM$(FUNIT,STATUS)
555   CALL EXIT
      END
```

Here is some sample output from a terminal session using the program just listed:

```
OK, SEG #ADD
ENTER PRIMARY KEY VALUE: 2194G
ANY SECONDARIES TO ADD FOR THIS RECORD? Y
ENTER INDEX NO.: 1
ENTER KEY VALUE: SPECTROGRAPHICS
MORE?Y
ENTER INDEX NO.: 2
ENTER KEY VALUE: NWOR
MORE?N
ENTER DATA RECORD (NON-KEY) INFO:  PORTLAND
READY TO QUIT? (Y OR N) N

ENTER PRIMARY KEY VALUE: 4056S
ANY SECONDARIES TO ADD FOR THIS RECORD? Y
ENTER INDEX NO.: 1
ENTER KEY VALUE: EASTERN GRAPHICS
MORE?Y
ENTER INDEX NO.: 2
ENTER KEY VALUE: NECN
MORE?N
ENTER DATA RECORD (NON-KEY) INFO:  OLD SAYBROOK
```

It is possible to add just primary key and data entries to the file using this program. Secondary index entries can then be added at later time, but the program would have to be modified to allow a FIND$ operation on a primary or secondary key value first, with FL$RET set. Then the array could be used on an ADD1$ call to add the secondary index entry.

Note

Since primary index and data entries are added separately from secondary index entries, there is no guarantee that a record for which you are in the process of adding secondary index entries cannot be deleted by another user before the current ADD1$ operation is complete. Files which are prone to multi-user access should be protected in some way, perhaps by another "single-threading" mechanism, much like that enforced by the MIDAS lock.

Array Word Values

The MIDAS array may return one of the following codes or values in word 1 after a call to ADD1$:

Value/Code	Meaning
Ø	Successful completion of call
1	Successful completion: there are duplicates for this index (okay)
7	No entry exists with supplied primary index value
12	Attempt to add duplicates to a primary index or to a secondary index that doesn't allow them
Others	See Appendix A for a list of MIDAS error codes

Direct Access Adds

In direct access files, the user must set the index value to -1 in all calls to ADD1$ that add data entries. To add secondary index entries, set index as described above. This indicates to MIDAS that the file is configured for direct access and that the user will be supplying a unique floating-point record number for each record added, in addition to a primary key value. The record number and the primary key can be the same, if desired (it's just a bit redundant). Remember that the record number does not have to be defined as a key during template creation because MIDAS takes care of storing the numbers.

In each call to ADD1$, the user must provide:

- A primary key value (in key)
- A floating-point data entry number (in words 3 and 4 of the array)
- The data entry size (in datasiz)

The data entry size is equal to the key length (rounded up, in words) plus the data length (in words) plus 2. Make sure you supply the correct data entry size every time.

The Array: The array format for direct access calls is shown earlier in Table 6-2. The contents of the first four words of the array in direct access calls to ADD1$ are:

- Word 1: Condition code (Ø or 1)
- Word 2: Data entry size (key size + data length + 2)
- Words 3-4: Entry number (record number) in REAL*4 format

If an entry already exists with the supplied record number, MIDAS places the new record into an overflow area. This will not happen however, if the primary key is defined as the record number because no duplicates are allowed for any primary key.

READING A MIDAS FILE

There is no "read" subroutine in the MIDAS-FORTRAN call interface, but FIND$ and NEXT$ provide the same service. At the FORTRAN call level, there are three types of read operations possible:

1. Keyed reads, by primary or secondary key, or by record number using FIND$ and NEXT$
2. Sequential reads, by primary or secondary key, or by record number -- using NEXT$
3. Sequential data subfile reads, using GDATA$ to return records in physical order stored

These three types of reads are discussed relative to the FIND$, NEXT$ and GDATA$ subroutines. Conceptually, FIND$ performs keyed or random reads. NEXT$ can do both keyed and sequential reads, while GDATA$ performs only sequential reads.

Note

Some of the other language interfaces automatically lock a record upon reading it. However, in the FORTRAN interface, none of the data retrieval routines locks a record upon positioning to it. The only way to explicitly lock a record is to use LOCK$. This makes sense because normally you only lock a record when you intend to update it.

FIND$

FIND$ locates and reads a MIDAS data entry by either primary or secondary key. Searches can be done on partial primary or secondary key values. The full key value as stored in the index subfile being searched, can be returned by FIND$ at the user's request. If a given secondary index contains secondary data, FIND$ can be told to return that instead of the data record.

FIND$ Calling Sequence

The calling sequence of FIND$ is:

```
CALL FIND$ (funit, buffer, key, array, flags, altrtn, index,
            file-no, bufsiz, keysiz)
```

The arguments which have special meanings in this call are shown in Table 6-6. Flag values for FIND$ are listed in Table 6-7.

Specifying Which Index to Use

The index argument indicates which access mode is being used on this call to FIND$. It also spells out which index is to be used in a keyed-index file. Settings are:

Index Values	Meaning
-1	Direct access -- locate entries by record number
Ø	Use primary index as search key
1 - 17	Use indicated secondary index as basis of search

See FIND$ and Direct Access for information on reading direct access files.

On pages 6-21 and 6-31, the argument file-no was mistakenly omitted from the argument explanation list. Please insert the following in between the index and bufsiz arguments on these pages:

file-no Set this to 0: obsolete.

Table 6-6. FIND$ Arguments

Argument	Meaning
funit	File unit number on which the MIDAS file is opened.
buffer	Buffer in which data entry, primary key values, or secondary data are returned as a result of call to FIND$.
key	User supplies full or partial primary or secondary key value to use on this call. In direct access, key is not supplied unless FL$UKY is set in flags: see below.
array	The communications array -- returned after each call with a completion or error code. In direct access, the user must supply the entry size and record number in words 2-4 of this array.
flags	See Table 6-7 for flags that can be used on calls to FIND$.
altrtn	Statement number in program of alternate return.
index	Set to Ø if primary index access. Set to 1-17 if secondary index access. Set to -1 for direct access.
bufsiz	Length of buffer; set to Ø if complete data entry and primary key (if FL$KEY set on) are to be returned; otherwise specify number of words you want returned from data subfile entry.
keysiz	Size of key; set to Ø if full key, otherwise set to number of bits, bytes or words in key.

Table 6-7. Flags for FIND$

Flag	Meaning
FL$BIT	When set on, length of key is specified in keysiz in bits, if the key is defined as a bit string, or in bytes, if the key is an ASCII string. When set off, keysiz is assumed to be in words.
FL$FST	Tells FIND$ to position to first entry in specified index subfile. If set off, FIND$ positions to first index entry that matches the user-supplied key, unless another flag setting overrules this.
FL$KEY	When set on, tells FIND$ to return the full value of the primary key for this record in buffer, beginning in buffer(1). The returned data record will then immediately follow the primary key in buffer.
FL$NXT	When set on, tells FIND$ to position to next index entry which is greater than current or user-supplied entry. When set off, positions to index entry that matches current or user-supplied entry; may be overruled by flag setting with higher precedence (FL$FST). This flag setting is useful in partial key searches.
FL$PLW	If set on, tells FIND$ to fetch the next entry in the current index, whether greater than or equal to ("not less than," for COBOL persons) the current or user-supplied entry. This is useful in partial key searches. When set off, positions to next entry in index only if it matches the current or user-supplied key value. (This can be overruled by other flag settings like FL$FST and FL$NXT.)
FL$PRE	When set on, tells FIND$ to position to index entry immediately prior to the current index entry. When set off, positions to index entry that matches the user-supplied key value, unless overruled by a flag setting with higher precedence.

FL$RET Tells MIDAS to return entire array after this call to FIND$. If set off, only the first word of array, the completion code, is returned.

FL$SEC When set on, returns secondary data from secondary index being searched instead of returning the data record. Secondary data is returned in buffer. Not applicable in direct access or primary index access, that is, if index is Ø or -1.

FL$UKY When set on, returns, in key, the full primary or secondary index value that corresponds to the user-supplied key value used in this call.

FL$USE In keyed-index access, when set on, tells MIDAS to use contents of array as returned by previous call. In direct access, the setting does not matter because the array is used anyway.

Specifying Key Values

If index is Ø, the user must supply a primary key value in the key argument. Similarly, if a secondary index is indicated, a value from that index must be specified in key so FIND$ can use it in the retrieval. If the full key value is specified in key, the keysiz argument can be set to Ø.

FIND$ Example

The following program reads records from the CUSTOMER file defined in Section 2. A variety of searches can be done with this program, including searches using full or partial primary or secondary key values. It also allows the user to retrieve the data subfile record referenced by the first entry in any specified index subfile, using the FL$FST flag. The program could easily be modified to find the previous or next index entry relative to the current entry by using the FL$PRE or FL$NXT flags. It is meant only as a general example of how to set up calls to FIND$, and is certainly not exhaustive.

```
C     READ PROGRAM FOR CUSTOMER FILE
C
C     THIS PROGRAM USES FIND$ TO LOCATE A RECORD BY ANY KEY
C     IT ALSO SHOWS HOW TO USE FL$FST TO RETURN FIRST ENTRY IN
C     ANY INDEX
C
$INSERT SYSCOM>KEYS.F
$INSERT SYSCOM>PARM.K
C     DECLARATIONS
C     Garden variety call parameters
      INTEGER*2 ARRAY(14),INDEX,FUNIT,BUFFER(35),STATUS,FLAGS
      INTEGER*2  KEY(13),BUFSIZ,KEYSIZ,CODE
C     KEY IS MAX OF 13 WORDS (SEC KEYØ1)
C
      INTEGER*2 ANSWER                          /* YES OR NO
      CALL OPENM$(K$READ+K$GETU,'CUSTOMER',8,FUNIT,STATUS)
      IF (STATUS .NE. Ø)  GO TO 1ØØ
Ø1    CALL TNOUA ('SEARCH ON PRIMARY? (Y or N)', 27)
      READ(1,3333)ANSWER
      IF(ANSWER .EQ. 'N') GO TO Ø5
C     ELSE IT'S A PRIMARY SEARCH
      INDEX = Ø
      GO TO 1Ø
Ø5    CALL TNOUA('ENTER SEARCH INDEX: ',2Ø) /* SEC. INDEX NO.
      READ(1,2222) INDEX
C     GO ON AND GET KEY VALUE
1Ø    CALL TNOUA('USE YOUR KEY VALUE? (Y OR N)',28)
      READ(1,3333)ANSWER
      IF(ANSWER .EQ. 'Y') GO TO 15
C     ELSE USE FL$FST FLAG TO GET FIRST INDEX ENTRY
      FLAGS = FL$FST + FL$RET
      GO TO 45
15    CALL TNOUA('ENTER KEY VALUE: ', 18)
```

```
      READ(1,1111)KEY
20    CALL TNOUA('PARTIAL KEY? (Y OR N)',21)
      READ(1,3333)ANSWER
      IF(ANSWER .EQ. 'Y') GO TO 30
C     ELSE ASSUME FULL KEY
      KEYSIZ = 0              /* FULL KEY
25    CALL TNOUA('RETURN ALL DATA? (Y or N)', 25)
      READ(1,3333)ANSWER
      IF (ANSWER .NE. 'Y') GO TO 35
C     OTHERWISE, SET BUFSIZ TO 0
      BUFSIZ = 0
      GO TO 40
30    CALL TNOUA('ENTER KEYSIZE IN WORDS: ', 24)
      READ(1,2222)KEYSIZ
      GO TO 25
35    CALL TNOUA('ENTER DATA SIZE: ', 17)
      READ(1,2222)BUFSIZ
      CONTINUE
C     MAKE CALL TO FIND$
40    FLAGS = FL$RET
45    CALL FIND$(FUNIT,BUFFER,KEY,ARRAY,FLAGS,$200,INDEX,0,
     + BUFSIZ,KEYSIZ)
C
C     DISPLAY WHAT'S IN BUFFER NOW
50    CALL TNOUA('RECORD READ IS: ',16)
      CALL TNOU(BUFFER, 70)
C     Ask if want to go on
60    CALL TNOUA('DO YOU WANT TO CONTINUE? (Y or N)', 33)
      READ(1,3333)ANSWER
      IF (ANSWER .EQ. 'N') GO TO 444
C     ELSE GO BACK TO TOP
      GO TO 01
C
C     FORMAT STATEMENTS HERE
1111  FORMAT(26A2)                          /* KEY VALUE
2222  FORMAT(I2)                            /* INDEX#,KEYSIZ,BUFSIZ
3333  FORMAT(A2)                            /* ANSWER
C
C     Error handlers next
100   CALL TNOUA('ERROR ON OPEN: STATUS IS: ',24)
      CALL TODEC(STATUS)
      GO TO 444
200   CALL TNOU('ERROR ON FIND: ',15)
      CALL TNOUA('INDEX IS: ',10)
      CALL TODEC (INDEX)
      CALL TONL
      CALL TNOUA('ARRAY(1) IS: ',13)
      CALL TODEC (ARRAY(1))
      CALL TONL
      GO TO 60
444   CALL CLOSM$(FUNIT,STATUS)
555   CALL EXIT
      END
```

Primary Key Search: The first example shows the use of a primary key to find a MIDAS file entry:

```
SEARCH ON PRIMARY? (Y or N)Y
USE YOUR KEY VALUE? (Y OR N)Y
ENTER KEY VALUE:  4Ø56S
PARTIAL KEY? (Y OR N)N
RETURN ALL DATA? (Y or N)Y
RECORD READ IS: 4Ø56S EASTERN GRAPHICS          NECN OLD SAYBROOK
```

Secondary Key Search: This sample excerpt shows a search conducted on a secondary key value.

```
SEARCH ON PRIMARY? (Y or N)N
ENTER SEARCH INDEX: 2
USE YOUR KEY VALUE? (Y OR N)Y
ENTER KEY VALUE:  NWOR
PARTIAL KEY? (Y OR N)N
RETURN ALL DATA? (Y or N)Y
RECORD READ IS: 2194G SPECTROGRAPHICS           NWOR PORTLAND
```

Using FL$FST: The FL$FST flag causes MIDAS to position to the first entry in the index specified and to return the associated data record. No key value need be supplied when FL$FST is used in a call to FIND$.

```
SEARCH ON PRIMARY? (Y or N)Y
USE YOUR KEY VALUE? (Y OR N)N
RECORD READ IS: Ø816S MORROW PAPER MILLS        NENH MONADNOCK
DO YOU WANT TO CONTINUE? (Y or N)Y
SEARCH ON PRIMARY? (Y or N)N
ENTER SEARCH INDEX: 1
USE YOUR KEY VALUE? (Y OR N)N
RECORD READ IS: 4Ø56S EASTERN GRAPHICS          NECN OLD SAYBROOK
```

Partial Key Values: Full or partial keys may be used in both primary and secondary key searches. Partial key values must be prefixes of the full key value -- that is, they must be taken from the beginning of the key value, not from the middle or the end. For example, if the key is "Massachusetts", legal partial values would be: "Mass", "Ma", "Massach", "M" and so forth. During a partial key search, FIND$ returns the first data entry that has a key value beginning with the indicated partial value.

To specify a partial key value, supply, in keysiz, the exact length of the value in key. If the key is a bit string or an ASCII string, its length should be specified in bits or bytes, as appropriate, and the FL$BIT flag should be set on. If FL$BIT is set off, the key size is assumed to be in words.

Here is an example of a partial key search using the READ program listed earlier:

```
SEARCH ON PRIMARY? (Y or N)N
ENTER SEARCH INDEX: 1
USE YOUR KEY VALUE? (Y OR N)Y
ENTER KEY VALUE:  EAST
PARTIAL KEY? (Y OR N)Y
ENTER KEYSIZE IN WORDS: 2
RETURN ALL DATA? (Y or N)Y
RECORD READ IS: 4Ø56S EASTERN GRAPHICS          NECN OLD SAYBROOK
```

If keys are not being stored in the record, set FL$KEY on in flags when doing partial key searches if you want the full key value to be returned.

Retrieval Options

FIND$ permits the retrieval of the following items from a MIDAS file:

- A data subfile entry (full or partial) -- set bufsiz to Ø to return all information. To return a partial entry, specify in bufsiz the number of words to be returned.

- The primary key value associated with the record being sought -- set FL$KEY on. This should be used when keys aren't stored in the data record, or when entries were added with FL$KEY set on during calls to ADD1$. Primary key value is returned in buffer. Set bufsiz to include both primary key and data record.

- The secondary data stored with the secondary key value on which the search is being conducted: secondary data is returned in buffer in place of data record. bufsiz can be set to Ø to return all secomdary data, or to the number of words you want returned. Use FL$SEC in the call: index must be a value from 1 - 17.

- A full primary or secondary key value when searching on partial keys -- set FL$UKY on: full key value is returned in key.

All information returned by FIND$ is placed in buffer (except when FL$UKY is used). Therefore, bufsiz must be specified accordingly during each call.

When to Use FL$KEY: The only cases in which it is necessary to set FL$KEY on (to return full primary key value) are: when you aren't storing keys in the data record, or when FL$KEY was set on during calls to ADD1$. In these cases, setting FL$KEY in a FIND$ is the only way you can find out what the primary key value is for a given record.

When to Use FL$UKY: The FL$UKY flag is useful when doing record access by partial key. It returns the complete value of the key which MIDAS used to conduct the search. If keys are being stored in the record, this is good way of checking that the index entries correspond to the key values in the data record. In addition, these flags come in handy when doing retrievals on duplicate keys as they can help identify which record you're actually looking at.

FIND$ and the Array

During access to keyed-index MIDAS files, the user doesn't have to worry about the settings for the array argument in calls to FIND$. Word 1 always returns a completion code to the user following a call to FIND$. If the value of array(1) is Ø, the call was successful. If 1, the key value used in the call may occur more than once in the file. If array(1) has a value other than Ø or 1, an error has occurred. Consult Appendix A to determine the nature of the problem.

The entire array is returned to the user when the FL$RET flag is set in the call to FIND$. It can subsequently be used in calls to other routines like ADD1$, NEXT$, DELET$ and LOCK$.

FIND$ and Direct Access

Direct access files can be accessed by any key or by entry number. To access a direct access file by primary or secondary use the keyed-index access method we've been talking about up to this point. Set index to the appropriate index subfile (key) number and set key and keysiz as described above. In other words, the direct access file is treated just like a keyed-index file. However, to access a direct access file by entry number (record number), index must have a value of -1, and both key and keysiz should be set to Ø. In some direct access files, the record entry number and the primary key may be the same, as in COBOL RELATIVE files.

Accessing a direct access file by entry number involves a search algorithm that calculates the physical location of the record in the file, given the entry number and the data subfile record size. To use the entry number method, supply a floating-point data entry number in array and the full data subfile entry size in bufsiz, in words.

Argument Settings: The index argument must be set to -1 in regardless of the search method used. If searching by primary key, key must contain the full primary key value to be used in the search, and keysiz must always be Ø, indicating a full key value. In the entry number access method, the array argument must be set to the entry number to be used on this call. See Table 6-2, Direct Access Array Format, earlier in this section. It doesn't matter whether or not you set FL$USE because the array is always used by MIDAS in this type of call.

NEXT$

NEXT$ is a powerful subroutine that allows a variety of operations to be performed on a keyed-index MIDAS file. NEXT$ can be used to:

1. Retrieve file records sequentially according to primary or secondary key order
2. Retrieve all file records with a primary or secondary key value greater than some given key value
3. Retrieve all records with the same partial key value
4. Retrieve all records with duplicate index entries for a specified secondary key value
5. Retrieve all records whose key values precede a certain key value in a given index subfile
6. Retrieve a particular record using a full or partial primary or secondary key value (keyed retrieval)

Special flag settings are required to perform these retrievals: see Table 6-9.

Note

NEXT$ cannot be used on direct access files. Therefore, index can never have a value of -1 in a call to NEXT$. Remember also that FL$RET must always be specified in calls to NEXT$, or a MIDAS error 3Ø will occur.

NEXT$ Calling Sequence

The calling sequence for NEXT$ is:

```
CALL NEXT$ (funit, buffer, key, array, flags, altrtn, index,
            file-no, bufsiz, keysiz)
```

Meanings for NEXT$ arguments are shown in Table 6-8; flags settings for NEXT$ are described in Table 6-9.

Table 6-8. NEXT$ Arguments

Argument	Meaning
funit	The file unit on which the MIDAS file is opened.
buffer	Buffer into which retrieved data record or secondary data value is read. If FL$KEY is set, buffer will include key value plus data record. If FL$SEC is set, secondary data is returned instead of data record. See Table 6-9.
key	Value of key to be used in search: can be full or partial, as specified in keysiz.
array	MIDAS communications array.
flags	For flag options to be used in this call, see Table 6-9.
altrtn	Statement number of alternate return to be taken in event of program error.
index	Indicates index to be used: if set to Ø, primary index is used; if greater than Ø, secondary index subfile is used. Direct access is illegal (index cannot be -1).
file-no	Set to Ø (obsolete).
bufsiz	Length of data to be returned: if set to Ø, full data subfile entry is returned. If FL$KEY is set on, the full key value is returned with the data. If FL$SEC is set on, secondary data is returned instead of the data subfile entry. Be sure to make the value of bufsiz large enough to accommodate everything that must be returned in buffer.
keysiz	Length of user-supplied key on this call. If set to Ø, full key value is used; if greater than Ø, partial key is specified in either bits or bytes (if FL$BIT is set on) or in words (FL$BIT set off).

Table 6-9. Flags for NEXT$

Flag	Meaning
FL$BIT	When set on, keysiz is specified in bits or bytes; when set off, keysiz is in words.
FL$FST	When set on, tells NEXT$ to return the record referenced by the first entry in the specified index.
FL$NXT	When set on, positions to next index entry greater than key.
FL$PLW	When set on, positions to next index entry greater than or equal to key.
FL$PRE	When set on, positions to index entry previous to current entry or user-supplied key.
FL$RET	Tells MIDAS to return contents of array after this call -- this flag must be set on in all calls to NEXT$, otherwise an error 30 will occur.
FL$SEC	When set on, returns secondary data in buffer instead of data record; only used when index is $\geq$ 1. If not set, secondary data is not returned.
FL$UKY	When set on, key field used on this call is updated by MIDAS with full key value stored in the index.
FL$USE	When set on, MIDAS uses current contents of array; array must be left over from a previous call to FIND$ or NEXT$.

Buffer Size Specifications

Data retrieved on a call to NEXT$ is returned in buffer. The amount to be read back is determined by the bufsiz argument. If the entire data subfile entry is to be returned, set bufsiz to Ø. Similarly, when retrieving secondary data (when index is set to a value greater than Ø and FL$SEC is set), set bufsiz to Ø. Otherwise, set this argument to the number of words you want returned from the index or data subfile. If FL$KEY is set in this call, the buffer size specified here is assumed to include the full primary key value. Make sure bufsiz specifies a large enough buffer to include the full primary key (as well as the data record) when FL$KEY is used.

Array Settings

The only word of the array likely to be of any concern to the user on calls to NEXT$ is word 1 which returns a completion code after the call. Again, the possible settings for array(1) are:

Code	Meaning
Ø	Successful retrieval
1	Successful retrieval, but duplicates may exist for this key value
anything else	Something is wrong: see Appendix A

Sequential Record Retrieval

To retrieve records sequentially from some point in a primary or secondary index (other than the beginning), use FIND$ to locate the initial key value. Once the starting point is found, repeated calls should then be made to NEXT$ to return the data subfile records based on the order of entries in the primary or in a secondary index. In combination, FIND$ and NEXT$ calls effectively enable a "greater than or equal to" search: first you find a particular value, then you find all the values that are greater than or equal to it. The key values are returned according to the collating sequence by which they were entered in the index subfile.

To start this type of retrieval, set the FL$RET flag in the call to FIND$ so the array can be used in the NEXT$ loop. Thereafter, set the FL$RET and FL$USE flags in each call to NEXT$ so the array can be used in that call and returned for use on the next call. (The FL$USE flag tells NEXT$ to use the array as returned by the previous call.)

From the Top: To retrieve file records sequentially, beginning with the first index entry in a given subfile, make a call to NEXT$ with FL$FST set on in flags. When set on, the FL$FST flag tells NEXT$ to return the record pointed to by the first index entry in the specified index subfile. The index to be used in the retrieval is specified in the index argument of the call. The FL$RET flag should also be set on in this call. After the initial call is made, the FL$FST flag is set off and the FL$PLW and FL$USE flags are set on in the subsequent call. This tells NEXT$ to get the next entry in the index regardless of whether it matches the one just retrieved or not.

NEXT$ Example

The program listed here accomplishes two types of retrievals: it retrieves all the entries in the CUSTOMER file, in order by primary index entry, and it finds all the duplicates of a particular secondary key value.

```
C     "NEXT" PROGRAM FOR CUSTOMER FILE
C
C     THIS PROGRAM RETRIEVES DUPLICATES FROM A SECONDARY INDEX
$INSERT SYSCOM>KEYS.F
$INSERT SYSCOM>PARM.K
C     DECLARATIONS
C     Garden variety call parameters
      INTEGER*2 ARRAY(14),INDEX,FUNIT,BUFFER(35),STATUS,FLAGS
      INTEGER*2  KEY(13),BUFSIZ,KEYSIZ,CODE
C     KEY IS MAX OF 13 WORDS (SECONDARY KEY Ø1)
      INTEGER*2 J                              /* TRACKS # OF DUPS
C
1Ø    CALL OPENM$(K$READ+K$GETU,'CUSTOMER',8,FUNIT,STATUS)
      IF(STATUS .NE. Ø) GO TO 2ØØ          /* ERROR HANDLER
C     GET ALL ENTRIES OUT OF PRIMARY INDEX
2Ø    INDEX = Ø
      KEYSIZ = Ø
      BUFSIZ = Ø
C
C     USE FL$FST FLAG TO GET FIRST ENTRY
3Ø    FLAGS = FL$RET + FL$FST
      CALL NEXT$(FUNIT,BUFFER,KEY,ARRAY,FLAGS,$3ØØ,INDEX,Ø,
     + BUFSIZ,KEYSIZ)
C
C     DISPLAY WHAT'S IN BUFFER NOW
      CALL TONL                              /* BLANK LINE
4Ø    CALL TNOUA('RECORD READ IS: ',16)
      CALL TNOU(BUFFER, 7Ø)
```

```
      CALL TONL

C
45    CALL TNOU('RETRIEVE RECORDS IN ORDER BY KEY', 32)
C
C     RETRIEVES RECORDS IN INDEX ENTRY ORDER
C
C     USING FL$NXT FLAG WHICH TELLS MIDAS TO
C     GET NEXT ENTRY GREATER THAN THE CURRENT
C     INDEX ENTRY.
C     DON'T FORGET TO RETURN ARRAY!!!
C
5Ø    FLAGS = FL$USE + FL$NXT + FL$RET
      CALL NEXT$(FUNIT,BUFFER,KEY,ARRAY,FLAGS,$6Ø,INDEX,Ø,
     + BUFSIZ,KEYSIZ)
      CALL TONL                              /* BLANK LINE
C
      CALL TNOU('NEXT RECORD IS: ',16)
      CALL TNOU(BUFFER,7Ø)
55    GO TO 5Ø                               /* REPEAT UNTIL DONE
C
6Ø    CALL TNOU('END OF ENTRIES IN INDEX Ø',25) /* END OF INDEX Ø
      CONTINUE
C
      CALL TONL
7Ø    CALL TNOU('FIND DUPLICATES IN SECONDARY INDEX Ø2', 37)
      INDEX = 2
      CALL TNOUA('KEY VALUE TO USE?', 17)
      READ(1,1111)KEY
C
75    FLAGS = FL$RET                         /* FIND FIRST ENTRY
      CALL NEXT$(FUNIT,BUFFER,KEY,ARRAY,FLAGS,$3ØØ,INDEX,Ø,
     + BUFSIZ,KEYSIZ)
C     PRINT OUT RECORD
      CALL TONL
      CALL TNOU('RECORD READ IS: ',16)
      CALL TNOU(BUFFER, 7Ø)
C
C     READ ALL DUPLICATES
C     J KEEPS TRACK OF # OF DUPLICATES FOUND
      J=Ø                                    /* INITIALIZE COUNTER
8Ø    FLAGS = FL$USE + FL$RET
      CALL NEXT$(FUNIT,BUFFER,KEY,ARRAY,FLAGS,$1ØØ,INDEX,Ø,
     + BUFSIZ,KEYSIZ)
C     PRINT OUT RECORD
      CALL TONL                              /* BLANK LINE
      CALL TNOU('NEXT DUPLICATE IS: ',19)
      CALL TNOU(BUFFER, 7Ø)
      J=J+1                                   /* KEEPS TRACK OF # DUPS
85    GO TO 8Ø
C
C     IF FALL THRU LOOP, CHECK VALUE OF J
1ØØ   IF (J .EQ. Ø) GO TO 15Ø                /* NO DUPS FOUND AT ALL
C     ELSE WE RAN OUT OF DUPLICATES
```

```
      CALL TNOU('NO MORE DUPLICATES',18)
      GO TO 444                              /* CLOSE FILE
C
150   CALL TNOU('NO DUPLICATES FOUND FOR THIS KEY', 32)
      GO TO 444                              /* CLOSE FILE
C
1111  FORMAT(26A2)                            /* FORMAT FOR KEY ARG
C     ERROR HANDLERS
C
200   CALL TNOUA('ERROR ON OPEN. STATUS IS: ',24)
      CALL TODEC(STATUS)
      CALL TONL
      GO TO 444
C
300   CALL TNOU('ERROR ON CALL TO NEXT: ',23)
      IF(ARRAY(1) .EQ. 7) CALL TNOU('RECORD NOT FOUND',16)
C     CLOSE FILE
444   CALL CLOSM$(FUNIT,STATUS)
555   CALL EXIT
      END
```

Sequential Retrieval Example: This sample session excerpt shows the records retrieved by the first part of the above program:

```
OK, SEG #NEXT

RECORD READ IS: 0816S MORROW PAPER MILLS        NENH MONADNOCK

RETRIEVE RECORDS IN ORDER BY KEY

NEXT RECORD IS:
1002P FLORA PORTRAITS           NENY CROTON-ON-HUDSON

NEXT RECORD IS:
2194G SPECTROGRAPHICS           NWOR PORTLAND

NEXT RECORD IS:
2334P PERFECT PRINTS            WRCA SANTA BARBARA

NEXT RECORD IS:
4056S EASTERN GRAPHICS          NECN OLD SAYBROOK

NEXT RECORD IS:
9402A ARTISTRY UNLTD.           WRCA MONTEREY

NEXT RECORD IS:
9411P STUDIO WEST               WRCA PALO ALTO
END OF ENTRIES IN INDEX 0
```

Retrieving Duplicates

NEXT$ can retrieve duplicate secondary key values and primary key values that begin with identical prefixes. Only prefix values can be used in partial key searches, that is, the key value supplied must be extracted from the initial portion of the full key value. For example, if the full key is "Brookline", legal prefixes would include "Brook", "Bro", "Br", and so forth.

To perform a duplicate key search, use a FIND$ (with FL$RET set), or use NEXT$ without FL$USE, to retrieve the first entry with the desired full or partial key value. The rest of the values that match this one can be found by calling NEXT$ with FL$USE set on. (FL$NXT and FL$PLN are set off.) The FL$RET flag should be set on for all calls to NEXT$ when doing this type of retrieval.

Duplicate Example: The NEXT program listed earlier allows you to retrieve records with duplicates of any secondary key value desired. For example, here is some output from a sample run of the NEXT program, using the value "WRCA" for secondary index Ø2:

```
FIND DUPLICATES IN SECONDARY INDEX Ø2
KEY VALUE TO USE?WRCA

RECORD READ IS:
9411P STUDIO WEST                WRCA PALO ALTO

NEXT DUPLICATE IS:
94Ø2A ARTISTRY UNLTD.            WRCA MONTEREY

NEXT DUPLICATE IS:
2334P PERFECT PRINTS             WRCA SANTA BARBARA
NO MORE DUPLICATES
```

Retrieving Previous Records

To find all the records whose index values are less than or equal to a certain key value, set the the FL$PRE flag on. First, you must have a key value to start with. Establish this with a call to FIND$ or NEXT$ with the FL$USE flag set off. Then call NEXT$ with FL$PRE, FL$USE and FL$RET set on. The search should fail when the first index entry in the index subfile is reached.

On page 6-47, the line describing bufsiz should read:

bufsiz Size of buffer in bytes.

(The manual currently says "words" instead of "bytes".)

GDATA$

The GDATA$ routine retrieves records directly from the data subfile in the physical order stored, that is, the order in which they appear in the data subfile. This order does not necessarily correspond to any key order, unless the records were added in order by primary key, or unless the file has just been MPACKed.

In order to use GDATA$, the FL$FST flag must be set on in the first call; the FL$NXT flag must be set on in subsequent calls.

Caution

Successive calls to GDATA$ with FL$NXT set should not be mixed with calls to other MIDAS file access routines because GDATA$ does not use the MIDAS array to keep track of current index position.

GDATA$ Calling Sequence

GDATA$ does not use the general calling sequence used by the other data access subroutines. Instead, it has its own:

```
CALL GDATA$ (unit, flags, buffer, bufsiz, status)
```

These arguments and their meanings are:

Argument	Meaning
unit	PRIMOS file unit on which MIDAS file is opened.
flags	Set FL$FST to retrieve first data record: this must be used for initial call. Set FL$NXT flag on to retrieve next sequential record in data subfile.
buffer	Buffer in which data is returned.
bufsiz	Size of buffer in bytes.
status	Error status code: may be one of the following:
	Ø No error
	>Ø System error code
	-1 Bad flags value supplied
	-2 Bad index descriptor
	-3 Invalid record position

The data buffer, buffer, contains the retrieved data record upon returning from a GDATA$ call.

UPDATING A RECORD

Record updates are performed by jointly using the LOCK$ and UPDAT$ routines. LOCK$ secures a record for update, preventing other users from locking or updating it. It does not, however, prevent other users from deleting a locked record. To update a record, it must first be locked with LOCK$ and then updated with UPDAT$ immediately thereafter.

MIDAS CALLS THAT PROCESS OTHER FILES CAN BE MADE BETWEEN CALLS TO LOCK$ AND UPDAT$

LOCK$

LOCK$ works on both keyed-index and direct access MIDAS files. It works essentially like FIND$ except that it also locks the record it retrieves. Like FIND$, LOCK$ returns the located data record in buffer. LOCK$ cannot lock an already locked record, and will return an error upon an attempt to do so. When LOCK$ is successful, the record will remain locked until UPDAT$ is called to update that record. A successful LOCK$ must always be followed by a call to UPDAT$ (otherwise the record can't be unlocked).

LOCK$ Calling Sequence

The LOCK$ calling sequence is:

```
CALL LOCK$ (funit, buffer, key, array, flags, altrtn, index,
            file-no, bufsiz, keysiz)
```

An explanation of these arguments appears in Table 6-10. Pertinent flags values are addressed in Table 6-11.

On page 6-48, add the following sentence to the end of the paragraph entitled UPDATING A RECORD:

> MIDAS calls that process other files can be made between calls to LOCK$ and UPDAT$.

On page 6-49, the argument listed as key value should be simply key.

Table 6-10. LOCK$ Arguments

Argument	Meaning
funit	The file unit on which the MIDAS file is open.
buffer	The buffer into which the data record to be locked is read.
key	Full primary or secondary key value which identifies the record to be locked; not necessary if record already found on a call to FIND$ or NEXT$.
array	For keyed-index access, set to 0 or 1 on input. For direct access, array must include the user-supplied record entry number and size to identify the record to be locked. See LOCK$ and Direct Access.
flags	For flags that can be used with LOCK$, see Table 6-11.
altrtn	An alternate return to be taken in the event of an error.
index	In keyed-index, set to 0 for primary index access, 1-17 for secondary index access, and to -1 for direct access.
file-no	Set to 0.
bufsiz	Length of data to be read from file: it's a good idea to set it at 0 (to return the whole record). However, if FL$KEY is set on, make sure bufsiz is large enough to include complete primary key.
keysiz	Ignored by LOCK$: full key, is assumed if supplied.

Table 6-11. Flags for LOCK$

Flag	Meaning
FL$KEY	Set on to include full primary key value in buffer along with data record; use only if keys are not stored in the record.
FL$RET	Must be set on in each call to LOCK$ so that UPDAT$ can use the array returned by this call. If not set, an error 3Ø occurs.
FL$USE	Set on if record to be locked was already found by an immediately prior call to FIND$ or NEXT$; the array returned by the previous call is used on this call to LOCK$, and the user does not have to supply a value for key.

Specifying a Key

In order to lock a record, it must be retrieved and made current. To retrieve the record to be updated, a full key value must be supplied in key on a call to LOCK$, or a valid array must be supplied by a previous call and FL$USE must be set on. The record to be updated can be positioned to and locked via the primary or secondary key. You may specify a value for keysiz if you want, but it will be ignored because a full key is always assumed if a value for key is supplied in the call. Partial key retrieval is possible only if you use FIND$ or NEXT$ first with FL$UKY and FL$RET set on in flags. A subsequent call can then be made to LOCK$ with FL$USE set on. No key is required in this call to LOCK$ because the data record has already been located by the previous FIND$ or NEXT$ call.

The Array in LOCK$

When the entry to be updated has already been found on a previous call to FIND$ or NEXT$ (with FL$RET set on), FL$USE should be set on in flags on the call to LOCK$. array can be set to 0 or 1 on the LOCK$ call. When returned, as a completion code, the first word of the array may contain one of the following values:

Value	Meaning
0	Successful retrieval
1	Successful retrieval but there may be duplicates of this key value (secondaries only)
7	Entry not found
10	Entry found, but already locked
Anything Else	See Appendix A

Note

On all calls to LOCK$, FL$RET must be set on so that the immediately subsequent call to UPDAT$ can use the array returned by the LOCK$ operation.

LOCK$ and Direct Access: In direct access, the use of LOCK$ is identical to its use on keyed-index access. The only differences are that index must be set to -1, and that array must include the data entry number and size on any call that does not use an array returned by a prior call to FIND$. The array must be set up as follows:

Word Number	Setting
1	If set to 1, tells MIDAS to use array contents. If set to -1, array contents are not used.
2	Supply entry size (in words). This includes the key length (in words) plus secondary data length (in words) plus 2 words.
3-4	Supply the record entry number. This is a single-precision (REAL*4) floating-point record number.
5-14	Set to Ø (obsolete).

Records may be retrieved prior to locking by calling FIND$ with FL$RET set on; LOCK$ is then called with FL$USE set on. There is no need to reset the array in this case.

UPDAT$

A call to UPDAT$ must always be preceded by a call to LOCK$. Check the returned completion code in array(1) after the LOCK$ call to make sure that the record was successfully locked before doing a call to UPDAT$. Record updates can be done on both keyed-index and direct access MIDAS files. An update is a true rewrite of the record as returned in buffer. After the call, the record is unlocked. A call to UPDAT$ with FL$ULK set on simply unlocks the locked record without updating it.

UPDAT$ Calling Sequence

The calling sequence for UPDAT$ is:

```
CALL UPDAT$ (funit, buffer, array, flags, altrtn, index,
             file-no, bufsiz, keysiz)
```

Because key values are not supplied in updates, both the key and keysiz arguments should be set to Ø in a call to UPDAT$. Index must match index specified on the prior call to LOCK$. The updated record is supplied in buffer. All the UPDAT$ arguments are described in Table 6-12. The flags that may be used on calls to UPDAT$ are described in Table 6-13.

Unlock Only: If an entry is to be unlocked only, (not updated) set FL$ULK on in flags. buffer does not have to be altered in this case.

UPDAT$ and the Array: Since a copy of the array as returned by LOCK$ must be used in every call to UPDAT$, the FL$USE flag must be set on. array is effectively supplied by FL$USE being set on and should not be tampered with following a call to LOCK$. The completion code returned

On pages 6-52 and 6-58, the key argument was inadvertently omitted from the calling sequences, although it does appear in the argument explanation lists on pages 6-53 and 6-59. The key argument should be inserted between the buffer and array arguments in these calling sequences.

On page 6-52, insert the key argument between the buffer and array arguments in the UPDAT$ calling sequence. The calling sequence should now read:

```
CALL UPDAT$ (funit, buffer, key, array, flags, altrtn, index,
             file-no, bufsiz, keysize)
```

Table 6-12. UPDAT$ Arguments

Argument	Meaning
funit	File unit on which MIDAS file is opened.
buffer	Buffer which contains the updated last record. If FL$KEY was set in the previous call to LOCK$, the primary key should be included in buffer. If keys are being stored in the data record, make sure all keys are supplied.
key	Ignored: set to Ø.
array	Supplied by previous call to LOCK$: should already be set to Ø or 1.
flags	For flags options on update calls, see Table 6-13.
altrtn	Alternate return to be taken in event of error.
index	Set to Ø if primary index was used in LOCK$ call; set to 1-17 if secondary index was used in LOCK$ call. If direct access, set to -1. index setting for UPDAT$ must match its setting in previous call to LOCK$.
file-no	Set to Ø.
bufsiz	Same as in call to LOCK$. (Ignored)
keysiz	Partial keys do not apply in updates. (Ignored)

Table 6-13. Flags for UPDAT$

Flag	Meaning
FL$KEY	When set on, indicates that a full primary key is present in buffer. Used only when keys are not being stored in the data records.
FL$ULK	When set on, tells UPDAT$ to unlock the record only: all other flags settings are ignored and the data record is not updated.
FL$USE	Must be set on so that the array returned by prior LOCK$ call is used on the UPDAT$ call. If not set on, an error 30 occurs.

FL$RET - see PTU 89

On page 6-54, add the following note to the bottom of Table 6-13:

FL$RET Ignored by UPDAT$. Note that UPDAT$ always resets the flag in the array which indicates whether or not a record is locked.

operation indicates whether or not the update was successful. If array(1) is returned as Ø, the update was successful. If returned as 11, the entry was not locked as expected and the operation failed. Other errors can occur as well, such as a concurrency error if the record was deleted by another user in between the LOCK$ and UPDAT$ calls. See Appendix A for a list of MIDAS error codes.

Caution

Neither primary nor secondary keys may be changed in a call to UPDAT$; this includes secondary data values. It's never permissible to change the value of a primary key anyway. Where keys are stored in the data record, changes to secondary key fields during a call to UPDAT$ will have no effect on the actual secondary index subfile entries that point to the record being updated. To change a secondary index entry and/or secondary data, you must delete the entry from the index subfile, and then re-add it in the desired form. If keys are being stored in the data record, you should then update the data subfile record accordingly.

UPDAT$ Example

The UPDATE program below shows the use of LOCK$ and UPDAT$ in updating a record in a MIDAS file.

```
C     UPDATE PROGRAM FOR CUSTOMER FILE
C
C     THIS PROGRAM FINDS A RECORD ON ANY INDEX
C     THEN LOCKS AND UPDATES IT AT THE USER'S REQUEST
C
$INSERT SYSCOM>KEYS.F
$INSERT SYSCOM>PARM.K
C     DECLARATIONS
C     Garden variety call parameters
      INTEGER*2 ARRAY(14),INDEX,FUNIT,BUFFER(35),STATUS,FLAGS
      INTEGER*2  KEY(13),BUFSIZ,KEYSIZ,CODE
C     KEY IS MAX OF 13 WORDS
      INTEGER*2 DATA(17)                        /* NON-KEY PART OF RECORD
C
      INTEGER*2 ANSWER                          /* YES OR NO
C
      EQUIVALENCE (BUFFER(19),DATA(1))
C
      CALL OPENM$(K$READ+K$GETU,'CUSTOMER',8,FUNIT,STATUS)
      IF (STATUS .NE. Ø)  GO TO 1ØØ
C
Ø1    CALL TNOUA ('SEARCH ON PRIMARY? (Y or N)', 27)
      READ(1,3333)ANSWER
      IF(ANSWER .EQ. 'N') GO TO Ø5
C     ELSE IT'S A PRIMARY SEARCH
      INDEX = Ø
```

```
      GO TO 10
05    CALL TNOUA('ENTER SEARCH INDEX: ',20) /* SEC. INDEX NO.
      READ(1,2222)INDEX
C     GO ON AND GET KEY VALUE
10    CALL TNOUA('USE YOUR KEY VALUE? (Y OR N)',28)
      READ(1,3333)ANSWER
      IF(ANSWER .EQ. 'Y') GO TO 15
C     ELSE USE FL$FST FLAG TO GET FIRST INDEX ENTRY
      FLAGS = FL$FST + FL$RET
      GO TO 25
15    CALL TNOUA('ENTER KEY VALUE: ', 18)
      READ(1,1111)KEY
      KEYSIZ = 0                          /* FULL KEY
      BUFSIZ = 0
C     MAKE CALL TO FIND$
20    FLAGS = FL$RET
25    CALL FIND$(FUNIT,BUFFER,KEY,ARRAY,FLAGS,$200,INDEX,0,
     + BUFSIZ,KEYSIZ)
C     ELSE PRINT OUT RECORD
C     DISPLAY WHAT'S IN BUFFER NOW
30    CALL TNOUA('RECORD READ IS: ',16)
      CALL TNOU(BUFFER, 70)
C
C     UPDATE RECORD
40    CALL TNOUA('UPDATE THIS RECORD?', 19)
      READ(1,1111)ANSWER
      IF(ANSWER .EQ. 'N') GO TO 01
C
C     ELSE LOCK AND UPDATE
      FLAGS = FL$USE + FL$RET
C     ALWAYS RETURN ARRAY ON CALL TO LOCK$
50    CALL LOCK$(FUNIT,BUFFER,KEY,ARRAY,FLAGS,$300,INDEX,0,
     + BUFSIZ,KEYSIZ)
C
C     IF NO ERROR, WE CAN UPDATE
C     CHANGE NON-KEY PORTION OF DATA RECORD
      CALL TNOUA('ENTER NEW DATA:',15)
      READ(1,4444)DATA
C
C     GO AHEAD AND UPDATE
      FLAGS = FL$USE
59    CALL UPDAT$(FUNIT,BUFFER,KEY,ARRAY,FLAGS,$400,INDEX,0,
     + BUFSIZ,KEYSIZ)
C
C     IF NO ERROR, ASK IF WANT TO CONTINUE
70    CALL TNOUA('DO YOU WANT TO CONTINUE? (Y or N)', 33)
      READ(1,3333)ANSWER
      IF (ANSWER .EQ. 'N') GO TO 444
C     ELSE GO BACK TO TOP
      GO TO 01
C
C     FORMAT STATEMENTS HERE
1111  FORMAT(26A2)                        /* KEY VALUE
2222  FORMAT(I2)                          /* INDEX#,KEYSIZ,BUFSIZ
```

```
3333  FORMAT(A2)                                  /* ANSWER
4444  FORMAT(34A2)                                /* DATA
C
C
C     Error handlers next
100   CALL TNOUA('ERROR ON OPEN: STATUS IS: ',24)
      CALL TODEC(STATUS)
      GO TO 444
200   CALL TNOU('ERROR ON FIND: ',15)
      GO TO 70
300   CALL TNOU('ERROR ON LOCK',13)
      CALL TNOUA('MIDAS ERROR IS:', 15)
      CALL TODEC(ARRAY(1))
      GO TO 444                                   /* CLOSE FILE
400   CALL TNOU('ERROR ON UPDATE',15)
      CALL TNOUA('MIDAS ERROR IS:', 15)
      CALL TODEC(ARRAY(1))
C     CLOSE FILE
C
444   CALL CLOSM$(FUNIT,STATUS)
555   CALL EXIT
      END
```

This excerpt from a sample execution of the above program shows a record update involving only the non-key portion of the data subfile record:

```
SEARCH ON PRIMARY? (Y or N)Y
USE YOUR KEY VALUE? (Y OR N)Y
ENTER KEY VALUE:  9402A
RECORD READ IS: 9402A ARTISTRY UNLTD.          WRCA MONTEREY
UPDATE THIS RECORD?Y
ENTER NEW DATA: CARMEL
DO YOU WANT TO CONTINUE? (Y or N)N
```

Now, use the READ program to see if the record was updated properly.

```
OK, SEG #READ
SEARCH ON PRIMARY? (Y or N)Y
USE YOUR KEY VALUE? (Y OR N)Y
ENTER KEY VALUE: 9402A
PARTIAL KEY? (Y OR N)N}
RETURN ALL DATA? (Y or N)Y
RECORD READ IS: 9402A ARTISTRY UNLTD.          WRCA CARMEL
```

DELET$

DELET$ can remove either a data subfile entry (and its associated primary key) or a secondary index entry. When a primary index entry and record entry are deleted, the associated secondary index entries (if there are any) are not deleted. Instead, they are marked for deletion, and the space they occupy is not reclaimed until the entry is referenced in an interface subroutine call, or until the file is MPACKed (see Section 12).

DELET$ ignores the fact that a record may be locked and deletes it anyway. It is not necesary to precede DELET$ with a call to another subroutine like FIND$ or NEXT$ because DELET$ can position to as well as remove a record or an index subfile entry. Records can be deleted from both keyed-index and direct access files.

DELET$ Calling Sequence

The DELET$ calling sequence is the same one shared by all the other interface routines. Some arguments are ignored by MIDAS and can be supplied as Ø in the call.

```
CALL DELET$(funit, buffer, array, flags, altrtn, index,
            file-no, bufsiz, keysiz)
```

The arguments for DELET$ are shown in Table 6-14.

Locating the Record to Delete

A record to be deleted may be located by either primary or secondary key value. Simply give DELET$ the full primary or secondary key value in key and set index appropriately. However, a FIND$, NEXT$ or LOCK$ operation may be used prior to a delete operation to establish the record to be deleted. In this case, the FL$RET flag should be set in this prior call so that DELET$ can use the returned array. The call to DELET$ must then set FL$USE in flags. The key and keysiz arguments are ignored when FL$USE is set on in the call.

Deleting Duplicates

When deleting duplicate key entries you must use NEXT$ to locate the particular record you want deleted. Neither DELET$ nor FIND$ will do the trick unless the record or index entry you want deleted happens to be the oldest of the duplicates in a particular subfile. The "oldest" duplicate is always the first one that physically appears in the index. Both DELET$ and FIND$ are unacceptable because they automatically position to the oldest duplicate value for that key in the file.

On pages 6-52 and 6-58, the key argument was inadvertently omitted from the calling sequences, although it does appear in the argument explanation lists on pages 6-53 and 6-59. The key argument should be inserted between the buffer and array arguments in these calling sequences.

Table 6-14. DELET$ Arguments and Flags

Argument	Meaning
funit	The file unit on which the MIDAS file is open.
buffer	Ignored.
key	Full primary or secondary key to be used in identifying the entry to be deleted. No need to supply a value for this if using the array from the previous call (assumes FL$USE is set).
array	Supply array(1) as Ø or 1 in keyed-index access. In direct access, words 2, 3 and 4 must include entry number and data size. See DELET$ and Direct Access.
flags	The only applicable flag in this call is FL$USE which should be set if a previous call to FIND$ or NEXT$ was made to locate entry to be deleted. All other settings for flags will be ignored.
altrtn	Alternate return to be taken if error on call.
index	Indicates whether a data record (and primary index entry) or a secondary index entry should be deleted. If Ø: deletes primary index entry and the data record it references. If 1-17: deletes secondary index entry from specified index. If -1: deletes primary index and data entry from a direct access file.
file-no	Set to Ø: obsolete.
bufsiz	Ignored on this call: set to Ø.
keysiz	Ignored on this call: set to Ø.

Deleting Secondary Index Entries

A secondary index entry may be removed without touching the data record it references, and without deleting the primary index entry associated with it. The entry to be deleted can be located by a call to DELET$ with FL$USE set off and index and key set to the index number and full key value to be used in the call.

Alternatively, the secondary index entry to be deleted can be located with a call to FIND$ or NEXT$ with FL$RET set in flags and with index set to the appropriate secondary index subfile number. The key value specified must be an entry within that index. A call to either FIND$ or NEXT$ is then followed by a call to DELET$ with FL$USE set on and with the value for index unchanged. key can be set to Ø since it is ignored when FL$USE is set anyway.

Removing a Record and All Keys

Although a records' secondary key entries will eventually be deleted, either by referencing them through a MIDAS call or by MPACKing the file, it is sometimes desirable to remove all secondary key entries explicitly. For example, in situations where a large number of records have been deleted and their secondary key entries have not, if someone did a call to FIND$ with FL$NXT set, for example, it would take quite a while for MIDAS to step through that secondary index looking for an index entry that points to an existing data record. Chances are that the secondary index subfile will contain many useless entries that point to nothing. One way to get around this is to delete all the secondary index entries before the actual data record is deleted.

First, delete all the secondary index entries that reference this record, then delete the data record and its primary index entry. This process is greatly simplified if you store all the keys in the data record, because once you retrieve the data record, you can tell what the keys are and you can delete the appropriate ones. If you don't make a habit of storing the keys in the data subfile record this could be a time-consuming process. In this case, you'd be better off deleting the record by primary key and then MPACKing the file to reclaim the secondary index and data subfile space.

DELET$ Example

Although it is not necessary to do a FIND$ before a call to DELET$, the program shown here does, simply because it's a good idea to verify that a record should be deleted before the delete actually occurs. This is true mainly in an interactive application, so you certainly can dispense with the precautions if you wish. This program also deletes all the secondary index entries associated with the record before the data entry is removed. This guarantees that no space will be taken up in any of the secondary index subfiles by useless entries.

```
C     DELETE PROGRAM FOR CUSTOMER FILE
C
C     THIS PROGRAM USES FIND$ TO LOCATE THE RECORD TO BE
C     DELETE BY PRIMARY KEY AND PRINTS IT OUT.
C     IT DELETES ALL SECONDARY INDEX ENTRIES FIRST, THEN
C     PRIMARY INDEX AND DATA SUBFILE ENTRY ARE REMOVED.
C
$INSERT SYSCOM>KEYS.F
$INSERT SYSCOM>PARM.K
C     DECLARATIONS
C     Garden variety call parameters
      INTEGER*2 ARRAY(14),INDEX,FUNIT,BUFFER(35),STATUS,FLAGS
      INTEGER*2 BUFSIZ,KEYSIZ,CODE
      INTEGER*2 PKEY(3),SKEY1(13),SKEY2(2)      /* KEYS FOR FILE
      INTEGER*2 ANSWER                          /* YES OR NO
C
C     EQUIVALENCE KEYS TO PARTS OF BUFFER
C
      EQUIVALENCE (PKEY(1),BUFFER(1)),
     * (SKEY1(1),BUFFER(4)),
     * (SKEY2(1),BUFFER(17))
C
10    CALL OPENM$(K$READ+K$GETU,'CUSTOMER',8,FUNIT,STATUS)
      IF (STATUS .NE. 0)  GO TO 100
15    CALL TNOUA('ENTER PRIMARY KEY VALUE OF RECORD TO BE DELETED:',48)
      READ(1,1111)PKEY
C     SET OTHER ARGUMENTS APPROPRIATELY
C
      INDEX = 0
      KEYSIZ = 0              /* FULL KEY
      BUFSIZ = 0
C     MAKE CALL TO FIND$
      FLAGS = FL$RET
20    CALL FIND$(FUNIT,BUFFER,PKEY,ARRAY,FLAGS,$200,INDEX,0,
     + BUFSIZ,KEYSIZ)
C     DISPLAY WHAT'S IN BUFFER NOW
25    CALL TNOUA('RECORD IS: ',11)
      CALL TNOU(BUFFER, 70)
C
      CALL TNOUA('OKAY TO DELETE?: ',17)
      READ(1,2222)ANSWER
      IF(ANSWER .EQ. 'N') GO TO 15
C     ELSE DELETE
C     NOW DELETE EACH SECONDARY INDEX BEFORE DELETING DATA RECORD
C
      INDEX = 1
      ARRAY(1) = 0                              /* RESET WORD 1 OF ARRAY
C     OTHER ARGS REMAIN UNCHANGED EXCEPT FOR KEY VALUE
30    CALL DELET$(FUNIT,BUFFER,SKEY1,ARRAY,FLAGS,$300,INDEX,0,
     + BUFSIZ,KEYSIZ)
C
C     NOW DELETE INDEX ENTRY FROM INDEX SUBFILE 02
      INDEX = 2
```

```
      ARRAY(1) = Ø
40    CALL DELET$(FUNIT,BUFFER,SKEY2,ARRAY,FLAGS,$3ØØ,INDEX,Ø,
     + BUFSIZ,KEYSIZ)
C
C     DELETE PRIMARY INDEX AND DATA ENTRY
      INDEX = Ø
      ARRAY(1) = Ø
5Ø    CALL DELET$(FUNIT,BUFFER,PKEY,ARRAY,FLAGS,$3ØØ,INDEX,Ø,
     + BUFSIZ,KEYSIZ)
C
6Ø    CALL TNOUA('ANY MORE ENTRIES TO DELETE?',27)
      READ(1,2222)ANSWER
C
      IF(ANSWER .EQ. 'Y') GO TO 15
C     ELSE CLOSE UP
      GO TO 444
C     FORMAT STATEMENTS
1111  FORMAT(6A2)                                  /* PKEY VALUE
2222  FORMAT(A2)                                   /* ANSWER
C
C     ERROR HANDLERS
1ØØ   CALL TNOUA('ERROR ON OPEN: STATUS IS: ',24)
      CALL TODEC(STATUS)
      GO TO 444
2ØØ   CALL TNOU('ERROR ON FIND: ',15)
      GO TO 15
3ØØ   CALL TNOU('ERROR ON DELETE',15)
      CALL TNOUA('MIDAS ERROR IS:', 15)
      CALL TODEC(ARRAY(1))
      GO TO 444                                  /* CLOSE FILE
C
444   CALL CLOSM$(FUNIT,STATUS)
555   CALL EXIT
      END
```

Below is some sample output from an execution of the above program:

```
OK, SEG #DELETE
ENTER PRIMARY KEY VALUE OF RECORD TO BE DELETED:9411P
RECORD IS: 9411P STUDIO WEST                WRCA PALO ALTO
OKAY TO DELETE?: Y
ANY MORE ENTRIES TO DELETE?N
```

To verify that the record is really gone, use the READ program:

```
OK, SEG #READ
SEARCH ON PRIMARY? (Y or N)Y
USE YOUR KEY VALUE? (Y OR N)Y
ENTER KEY VALUE:  9411P
PARTIAL KEY? (Y OR N)N
RETURN ALL DATA? (Y or N)Y
ERROR ON FIND:
INDEX IS:      Ø
ARRAY(1) IS:      7
DO YOU WANT TO CONTINUE? (Y or N)N
```

The error code 7 means that the record sought was not found, which was certainly not unexpected.

DELET$ and Direct Access

To delete a record from a direct access file, the user must supply a full primary key in key or a floating-point data entry number and data entry size in array.

The only way to delete a secondary index entry from a direct access file is to pretend it's not a direct access file and supply the proper index number in index instead of the usual value of -1.

SECTION 7

THE COBOL INTERFACE

INTRODUCTION

The COBOL interface to MIDAS is based on the standard COBOL file I/O statements for INDEXED SEQUENTIAL and RELATIVE files. Keyed-index MIDAS files are called INDEXED SEQUENTIAL files in COBOL and direct access MIDAS files are known as RELATIVE files. Any kind of MIDAS file can be accessed through the COBOL interface just as if it were any other standard COBOL INDEXED SEQUENTIAL or RELATIVE file. All the "housekeeping" details associated with the FORTRAN call level interface to MIDAS are built into both sets of COBOL file I/O statements, which means that there's no need for the user to worry about them. In addition, the COBOL file-handling packages come equipped with error-handlers that simplify routine file processing and debugging.

A template must be created with CREATK for both types of files. COBOL cannot "create" a MIDAS file from program level; it can only access an existing file. Any MIDAS file referred to in this section already exists; that is, a template has been created for it with CREATK.

What's in this Section

This section explains how to access an INDEXED SEQUENTIAL file from a COBOL program. It describes how to define the file's characteristics properly in the various parts of a typical COBOL program, and explains the syntax of the COBOL statements used to read, write and update records in a MIDAS file. Only INDEXED SEQUENTIAL MIDAS files are covered in this section; RELATIVE files are described in Section 11.

It is assumed that you have previously read Sections 2 and 3 and that you already know how to create a template for a keyed-index MIDAS file. The syntax and usage of the COBOL statements and verbs as they pertain to INDEXED SEQUENTIAL files are summarized briefly here; refer to The COBOL Reference Guide for more details on COBOL syntax and concepts.

This section and Section 11 supplement those portions of The COBOL Reference Guide that deal with MIDAS files. This book documents the latest version of MIDAS and therefore should be considered the more up-to-date of the two books. However, The COBOL Reference Guide should always be considered the authority on syntax, as this book tends to focus only on those aspects of COBOL that apply directly to MIDAS. The chief function of sections 7 and 11 is to summarize all the important tools which are needed in dealing with MIDAS files. Perhaps equally importantly, they show you how to use these tools in a straightforward and practical manner.

More Terms

The primary key in an INDEXED SEQUENTIAL file is called the RECORD KEY; secondary keys are called ALTERNATE RECORD KEYS. Prime's COBOL supports the use of up to five secondary keys in INDEXED SEQUENTIAL files. The primary key in a RELATIVE file is called the RELATIVE KEY and is always a record number. Details on RELATIVE file access are covered in Section 11.

Language Dependencies

While supporting most MIDAS features, COBOL places these limitations on MIDAS files used in COBOL applications:

- Up to five secondary keys are supported per file (INDEXED only).
- The primary key and all secondary keys (if any) must be included in the data record -- known as "storing keys in data".
- The primary key must be the first field in the data record.
- Secondary keys should not be embedded in the primary key because if you change any of the secondary key values, you will impact the primary key field which cannot be changed.
- If a MIDAS file has fixed-length records, the data size specified in the level Ø1 description of the data record must match the data size defined for the file during CREATK.
- The only key types supported by COBOL are ASCII (A) and bit string (B). The maximum ASCII key size is 64 characters or bytes (32 words); the maximum bit string key size is 32 characters (16 words). Avoid specifying any other type of key during template creation.
- Secondary data is not supported (see Appendix C).

Restrictions applicable to RELATIVE files are covered in Section 11.

Note

COBOL supports MIDAS files with either fixed- or variable-length records. However, even though the MIDAS file template may be declared with variable-length records, COBOL always writes out fixed-length records.

Compiling and Loading a COBOL Program

COBOL programs that access MIDAS files must be loaded with the MIDAS library, VKDALB. Here is an example of a typical compile and load sequence, showing all the libraries that must be loaded in order to successfully run a program:

```
COBOL program
SEG
VL #program
LO B_program
LI VCOBLB
LI VKDALB
LI
SA
Q
SEG #program
etc.
```

You should substitute the appropriate program name for the program parameter shown in the above sequence.

Example: This is an example of a compile, load and run sequence. User input is underlined.

```
OK, cobol read.da
 Phase I
 Phase II
 Phase III
 Phase IV
 Phase V
 Phase VI


No Errors, No Warnings, Prime V-Mode COBOL, Rev 17.6 <DACUST>
```

```
OK, seg
[SEG rev 17.6]
# vl #read.da
$ lo b_read.da
$ li vcoblb
$ li vkdalb
$ li
LOAD COMPLETE
$ sa
$ q

OK, seg #read.da
ENTER FILE ASSIGNMENTS:
> (enter filenames or / -- see below )
FILE ASSIGNMENTS COMPLETE.
(program now runs -- hopefully)
```

If the file-id values used in the program to identify existing disk files are not identical to the actual pathnames of the files (relative to the current directory), you must indicate which program file-id values correspond to which disk files, using the form:

```
file-id-value = actual-pathname
```

If the file-id-values are identical to the actual filenames of the files to be accessed, simply enter a "/" character in response to the "ENTER FILE ASSIGNMENTS" prompt as indicated above. For more information, see Data Division Requirements, below.

OPENING A MIDAS FILE

In order to open a MIDAS file from a COBOL program, the standard COBOL file I/O procedures must be followed. The rules for defining an INDEXED SEQUENTIAL file in the File Control Section and in the Data Division of a program are discussed below.

File Control Requirements

The File Control paragraph contains the names of the files to be accessed, the names of the devices on which they are to be opened, optional mode specifications, the names of the primary (RECORD) key (one for each file), the names of any secondary keys (ALTERNATE RECORD) present in each file and an optional file status indicator (one for each file) to be used in monitoring the success or failure of the various program statements that affect each file.

The basic format of the SELECT statement for an INDEXED file is:

```
SELECT filename

 ASSIGN TO PFMS

 ORGANIZATION IS INDEXED

  [ACCESS MODE IS  { SEQUENTIAL }
                   { RANDOM     }  ]
                   { DYNAMIC    }

 RECORD KEY IS key-name-1

 [ALTERNATE RECORD KEY IS key-name-2 [WITH DUPLICATES]...]

 [FILE STATUS IS status-code].
```

For a complete discussion of File Control Section rules, refer to The COBOL Reference Guide. The important rules are summarized below.

The SELECT Clause: Simply defines the name of the MIDAS file and tells the compiler to assign it some available file unit. Always assign the file to PFMS. Indexed files can only be accessed by MIDAS if they are disk files. Each filename specified in a SELECT statement must also appear in a Data Division FD entry. (See Data Division Requirements, below.)

The Organization Clause: Is probably the single most important part of the File Control definition sequence because it tells the compiler that the file to be opened is a keyed-index MIDAS file.

The ACCESS MODE Clause: Is optional; the default mode is SEQUENTIAL. If SEQUENTIAL is specified, or if the clause is omitted, all reads and writes must be performed sequentially -- no random operations are allowed. Records must be added in primary key order and are always retrieved in key order in SEQUENTIAL access mode.

If the RANDOM access mode is chosen, records can be written and retrieved in a random fashion, based on a supplied key value. Sequential reads and writes are not permitted. The DYNAMIC access mode gets you the best of both modes: you can read and write sequentially or randomly and you can switch back and forth between the two.

The RECORD KEY Clause: Defines the key-name associated with the primary key for the MIDAS file. The parameter key-name-1 must be defined in the Record Description entry associated with the FD entry for this file, and must be the first entry in such a description.

Some general rules of interest governing the RECORD KEY definition are:

- A primary key cannot be specified with an OCCURS clauses.
- No duplicate entries are allowed for the primary key.
- The length of the primary key cannot exceed 64 characters if it's an ASCII key or 32 characters if it's a bit string. It must be the same length and type as defined during template creation.
- The primary key cannot have a P character in its PICTURE clause, nor can it be defined as a numeric with a separate sign.
- The primary key should not have secondary keys embedded within it because the primary key value cannot be changed. Since secondary key values can be altered, it is too easy to get into trouble by embedding keys in this manner. If you do it, use extreme caution.
- Keys must not be defined in the WORKING STORAGE area of the program.

When defining the keys in the File Control section, if you're not sure of any of the length or types, use the PRINT option of CREATK to obtain a summary of each index description. See Sections 2 or 12 for more information.

The ALTERNATE RECORD KEY Clause: Designates a field within each record as a secondary key. Recall from Sections 1 and 2 that a MIDAS file may have secondary keys that are specified during template creation. In COBOL, you must always define the keys in the order in which they were created during CREATK, that is, index Ø, index 1 ... and so on. The ALTERNATE RECORD clause tells COBOL about the order and length of each field you've designated as a key for this MIDAS file. A separate ALTERNATE RECORD KEY clause is required for each secondary key you've defined in the template. The WITH DUPLICATES modifier should be used only for those keys which were allowed duplicate status during CREATK and is a documentation feature only; you cannot alter the duplicate status of an index at program level. This can only be done with CREATK; see Section 12 for more information.

Secondary keys are bound by the same size and type restrictions as the primary key and the cannot have P characters in their PICTURE clauses. Secondary keys cannot be defined in the WORKING STORAGE area of the program either.

The FILE STATUS Clause: Names a two-byte unsigned field, called status-code in the format above. The COBOL file control system uses it to indicate the execution status of each program statement which explicitly references the MIDAS file. Each time one of these statements is executed, a 2-byte status code is placed into this field indicating whether or not the operation was successful. Each status code describes a different condition or problem, as shown in Table 7-1.

Table 7-1. Status Codes for Indexed Files

Status Code	Error Type	Interpretation
ØØ	NONE	Successful completion of operation.
1Ø	END OF FILE	End of file reached on READ operation; file pointer positioned past logical end of file.
22	INVALID KEY	Attempt to perform a WRITE or REWRITE which would create a duplicate primary key entry. Duplicate primary key values are illegal.
23	INVALID KEY	Record not found on an unsuccessful key search; there is no record in the file with this key value.
3Ø	SYSTEM I/O ERROR	Operation unsuccessful due to an I/O error, such as a data check, parity error or a transmission error.
9Ø	Prime-defined Conditions are codes 9Ø-99	Record already locked; another user or process has already locked this record for update.
91		Record not locked; a REWRITE operation was attempted without first locking the record via a READ operation.
92		Attempt to add a duplicate secondary key value to a secondary index subfile that does not permit duplicates.
93		The indexes referred to in the program do not match those defined during template creation.
94		MIDAS concurrency error: another user just deleted the record you were trying to perform some operation on.
95		Bad record length supplied: the program has incorrectly specified the record length (data size) of the MIDAS file.
99	SYSTEM ERROR	System error: could be serious trouble. Verify that the program is not seriously flawed before you call your system analyst.

Example: Below is a sample File Control Section from a program used to access the CUSTOMER file created in Section 2. Consult the sample program at the end of this section for another example of a File Control Section.

```
FILE-CONTROL.
    SELECT CUSTOMER ASSIGN TO PFMS
     ORGANIZATION IS INDEXED
     ACCESS MODE IS SEQUENTIAL
     RECORD KEY IS CUST-ID
     ALTERNATE RECORD KEY IS CUST-NAME WITH DUPLICATES
     ALTERNATE RECORD KEY IS LOCATION-CODE WITH DUPLICATES
     FILE STATUS IS STATUS-CODE.
DATA DIVISION.
```

Data Division Requirements

The Data Division of the program describes the record structure of each file used in the program, that is, each file mentioned in the File Section. It also describes the record structure and data items which are not part of external files but which are used to handle data written to and read from these files during the course of program execution.

The important rules governing the File Section of the Data Division are summarized below:

The File Section of the Data Division consists of the following:

- One or more file description entries called FD's. If you have more than one FD, the keys must be specified in the same order in each one. Further, the order in which the keys are described in each FD should match the order in which they appear in the record.
- One or more record description entries for each FD.
- One or more file-id values, which name the MIDAS files referred to in the File Control Section. The file-id value may or may not be the same as the filename specified in the File Control Section.

The general format of a File Description entry is:

```
FD filename [UNCOMPRESSED]

LABEL { RECORD IS  } STANDARD
      { RECORDS ARE}

[RECORD CONTAINS integer-2 [TO integer-3] CHARACTERS]

VALUE OF FILE-ID IS file-id-value

[DATA { RECORD IS  } data-name-1 [data-name-2] ...].
      { RECORDS ARE}
```

Here are some important points about the File Section clauses:

- The FD clause, required for a MIDAS file, must be followed by the name which the program uses to refer to this MIDAS file. The UNCOMPRESSED option may be specified, but is ignored by MIDAS.

- The LABEL RECORD IS STANDARD clause is required for all disk files.

- If the RECORD CONTAINS clause is used, the number of characters specified must match the data record size specified during template creation. The maximum record size is 32767 characters.

- If the DATA RECORD clause is used, a record description must follow this clause. If more than one DATA RECORD is defined per file, a separate description of each one must be given. The beginning of each new record description must begin with an Ø1 level number. Multiple record descriptions imply that a file has more than one record-type -- but all share the same buffer area. The key fields must all be specified in the same relative order in each DATA RECORD description. This order must correspond to the order in which the keys were defined during template creation.

- The VALUE OF FILE-ID clause is required; the value of file-id-value is used to tie an internal filename to the actual name of the MIDAS file as it appears on disk. See Note below.

Note

Since COBOL only accepts file-id values of 8 characters or less, the file-id feature should be used to assign another name to any MIDAS file whose pathname exceeds 8 characters. The actual MIDAS pathname is equated to the internal filename (which is specified as file-id-value) at run-time. When COBOL asks for FILE ASSIGNMENTS during run-time, the actual MIDAS filename and the file-id value are equated in the form:

file-id-value = actual-MIDAS-file-pathname

The actual MIDAS filename may be assigned to file-id-value in the VALUE OF FILE-ID clause if it's less than 8 characters.

The Record Description simply defines all the items that make up a record and their relationship to one another. The complete syntax of a Record Description entry is described in the Data Division chapter of The COBOL Reference Guide.

Example: This example shows a sample File Section in the DATA DIVISION of the program whose Environment Division appeared earlier:

```
DATA DIVISION.
FILE SECTION.
FD  CUSTOMER
    LABEL RECORDS ARE STANDARD
    VALUE OF FILE-ID IS "CUSTOMER"
    DATA RECORD IS CUST-FILE-RECORD.
Ø1  CUST-FILE-RECORD.
    Ø2  CUST-ID PIC X(5).
    Ø2  CUST-NAME  PIC X(25).
    Ø2  LOCATION-CODE.
        Ø5  REGION      PIC XX.
        Ø5  STATE       PIC XX.
```

The OPEN Statement

The OPEN statement opens the MIDAS file and establishes the mode in which it is to be accessed. It must be executed before any other statement that references the file. More than one file can be opened with this statement, but each file name specified in an OPEN statement must appear in a SELECT and ASSIGN statement and must be described with an FD entry in the Data Division. The format is:

$$\text{OPEN} \left\{ \begin{matrix} \text{INPUT} \\ \text{I-O} \\ \text{OUTPUT} \end{matrix} \right\} \text{filename}$$

The filename is the internal name for the MIDAS file; INPUT, OUTPUT and I-O are the access modes which may be applied to the file. (See Access Modes below for details.)

If the named file cannot be located, the program will abort at run-time. This statement may be used to open more than one file as shown in this example:

```
OPEN INPUT CARD-FILE
     OUTPUT PRINT-FILE, DIRECTORY-FILE.
```

Access Modes: Briefly, here's what you can do in each access mode:

- In INPUT mode, a file can only be accessed by READ statements ("read only" mode).
- In OUTPUT mode, a file can only be written to with WRITE statements ("write only" mode).
- In I-O mode, a file can be both read and written to, and records can be updated and deleted as well. Records are automatically locked when read in I-O mode, whereas they are not in INPUT mode. (See Record Locking later in this section.)
- Table 7-2 shows what statements can be used in each access mode.

The CLOSE Statement

The CLOSE statement is the reverse of the OPEN statement: it tells the compiler that the file unit on which the file is opened should be released and that the file should be written back to disk in its current form. The format is simply:

```
CLOSE filename
```

filename is the name of the file as specified in the SELECT and FD clauses. A file can be OPENed and CLOSEd more than once in the same program.

Table 7-2. Statements Permitted in Each Access Mode

INDEXED I/O

File Access Mode	Statement	Open Mode		
		Input	Output	Input-Output
Sequential	READ	●		●
	WRITE		●	
	REWRITE			●
	START	●		●
	DELETE			●
Random	READ	●		●
	WRITE		●	●
	REWRITE			●
	START			
	DELETE			●
Dynamic	READ	●		●
	WRITE		●	●
	REWRITE			●
	START	●		●
	DELETE			●

ERROR HANDLING

Run-time errors can be handled in a number of ways. Among them, those well-suited to MIDAS file processing are:

- The AT END clause, which directs control to some "end-of-file" handler (procedure) in the event that a logical end of file is reached during a read.
- The INVALID KEY clause, which is executed when an error occurs; it transfers program control to a designated procedure or performs some useful action or series of actions. (Can be used in START, READ, WRITE, REWRITE and DELETE statements.)
- The USE AFTER ERROR statement which indicates the name of a procedure in the program which will be executed in the event of an I-O error, in addition to the System's standard I-O procedures.

A brief explanation of how these error-handlers work follows; refer to The COBOL Reference Guide for complete details on these error handling mechanisms.

The AT END Clause

The AT END clause, used only in a sequential READ statement, prevents program failure when an end-of-file condition is encountered during the read. The format is simple:

```
AT END imperative-statement
```

The imperative-statement, for example, could be a GO TO that transfers control to another procedure in the program which performs some further useful action, or it might simply close the file. Of course, this statement could be any number of things; these are just a few suggestions.

The INVALID KEY Clause

Many things can go wrong in MIDAS file processing, as evidenced by Table 7-1. The INVALID Key clause is one good way to identify and trap these errors. It should be used in all START, READ, WRITE, REWRITE, and DELETE statements to protect your program from key errors.

Without an INVALID Key clause or a USE AFTER statement, the program will abort when a file I-O error occurs. The format of the clause, which can be appended to any of the statements mentioned above is:

```
INVALID [KEY] imperative-statement
```

The word "KEY" is optional: INVALID is sufficient. When this clause is executed, the cause of the error may be determined by examining the FILE-STATUS variable. For a list of codes, see Table 7-1.

For example, this READ statement is "protected" by an INVALID KEY clause:

```
READ MFILE KEY IS PKEY INVALID KEY PERFORM READ-ERROR.
```

If a key error in the MIDAS file MFILE is raised during this READ, a procedure named READ-ERROR will be performed.

The USE AFTER Statement

The USE AFTER statement must be placed under the DECLARATIVES section of the program, immediately following a section header (and a period and a space). It defines a procedure which will be executed in addition to the standard I-O control system's method of dealing with I-O errors.

The format of USE AFTER is:

```
USE AFTER [STANDARD] {EXCEPTION} PROCEDURE ON {filename}
                     {ERROR    }              {INPUT   }
                                              {OUTPUT  }
                                              {I-O     }
```

The INPUT, OUTPUT, I-O, and filename parameters are used to indicate when that particular procedure should be executed. When filename is specified, only errors occurring while processing that file will be handled by this procedure. This is useful for multi-file processing. USE AFTER itself is never executed: instead, it identifies the conditions under which the procedure it introduces should be executed. The terms EXCEPTION and ERROR are synonymous.

```
PROCEDURE DIVISION.
DECLARATIVES.

Section-name SECTION.  USE AFTER etc.
[paragraph-name.  [sentence] ...]...
```

After the execution of the USE procedure, program control is returned to the statement following the invoking statement. This example shows a simple USE procedure:

```
PROCEDURE DIVISION.
DECLARATIVES.
ERROR-HANDLING SECTION.  USE AFTER ERROR PROCEDURE ON I-O.
 READ-ERR.
   DISPLAY 'STATUS CODE IS:' ERROR-STATUS.
     .
     .
     .
    etc.
END DECLARATIVES.
```

You can have a separate USE AFTER procedure for each file accessed in the program: or, you can have a single procedure for each OPEN mode, that is, one for INPUT errors, one for OUTPUT errors, and one for I-O errors (covers everything). Alternatively, you can have a single USE AFTER that handles all the errors that might occur on any of the files accessed by the program.

POSITIONING THE FILE

The concepts of "file position" and the "current record" are very important when dealing with MIDAS files. It's a good idea to understand them before you attempt to figure out how the actual I/O statements work -- it will spare you a lot of confusion later on.

A Logical View

From the user's point of view, "file position" refers to the file pointer's present position in the file. The record to which it is currently pointing is the "current record". The COBOL statements which alter the current record location are OPEN, START, READ and DELETE. DELETE alters the current record pointer by leaving the current record position undefined after a record is deleted. However, this is no problem because it's still possible to do a sequential or keyed READ after a DELETE.

What's Really Happening

The first and most important thing to understand is that file positioning is actually done relative to a primary or secondary index subfile. The user sees file positioning in terms of the data subfile, that is, in terms of which record is returned at any given point; however, the file position is actually based on which key the user supplies in a given START or READ statement. If you specify that a START should be done using the primary key (the RECORD KEY), file position will be established via the primary index subfile. If the file is then processed sequentially, data subfile records will be returned in primary key order; in other words, MIDAS uses the order of entries in the primary index subfile as a basis for finding and returning data subfile records.

Similarly, the secondary index subfile can be used as a basis for file processing if a secondary key value is used in a START or a keyed READ. A sequential read thereafter would return the next data subfile record as referenced by the next sequential entry in the secondary index subfile. Remember that MIDAS always adds entries to the index subfiles in sorted order; that is, it inserts things where you would logically expect them to be inserted. That's why it always seems that records are in some logical order, when they are really just stored in the order in which they were written.

Record Locking

Record locking is another important concept in MIDAS that applies to files opened for I-O only. To protect users from conflicting updates and to ensure that the record you just read will be the record you update or delete (assuming you want to perform either of these operations), the READ statement always locks the record to which it positions. Remember, this happens only in files opened for I-O. When a record is "locked", it becomes, in a sense, the sole property of the user who locked it, protecting the record from harm by any other user as long as the record remains locked. This is a general rule, and of course, there are always exceptions. (One such exeception is described under START and Locked Records, below.) The record remains "locked" until another I/O operation is performed. Only the "current record" can be locked and READ is the only COBOL statement that can lock a record. There are no specific "lock" and "unlock" statements in COBOL.

What If Somebody Else...: If your program attempts to access a record which someone else has already locked, a MIDAS error will occur. If you don't have a "program trap" for this condition, the code of a MIDAS error message will appear at your terminal and the program will abort. In addition, a file status code of 9Ø will be returned. To be safe, make sure that your program handles all the file status conditions listed under File Status codes in Table 7-1. For details, see ERROR HANDLING, above.

The START Statement

The START statement establishes file position in a MIDAS file opened for SEQUENTIAL or DYNAMIC access by moving the file pointer to a specific record in the file. START cannot be used in a file opened for RANDOM access.

Generally, START uses a specific key value or a conditional expression based on a key value to position a MIDAS file to a particular record. If an explicit key value is not supplied by the program, the primary key (the RECORD KEY) is assumed. To position the file on an explicit key value, follow these two steps:

1. Use a MOVE statement to assign some initial value to the key you want to use in the START operation.

2. Use a START statement to specify whether the file should be positioned to the first record containing that key value or to the first record with a key value greater than the value just MOVEd to that key, or to the first record with a key value greater than or equal to the specified key value.

The general format of START is:

```
START filename [KEY IS { GREATER THAN          } key-name]
                       { NOT LESS THAN         }
                       { EQUAL TO              }

[INVALID KEY imperative-statement].
```

key-name is the name of a file key (which must have been defined in the RECORD definition) and contains some value previously assigned by a MOVE operation. The value in key-name is used for comparison by the START statement. The INVALID KEY clause must be included unless a USE AFTER procedure has been provided for this file under the DECLARATIVES.

Important Points: The important points to note about START are:

- START only positions the file pointer -- it does not return the record (as in a READ).

- START can only be used with files opened for SEQUENTIAL or DYNAMIC access.

- If the key value specified in the previous MOVE does not exist, the program will terminate abnormally unless some error-handling mechanism is included in the program. See ERROR HANDLING above.

- Both primary and secondary key values can be used to position the file, but a value must be assigned either to the RECORD KEY (primary key) or to an ALTERNATE RECORD KEY (a secondary key) prior to the START statement. If a secondary key is used, the KEY IS key-name clause must be included in the START statement.

- The GREATER THAN option positions to the first file record whose key value is greater than that assigned to key-name.

- The NOT LESS THAN option positions to the first record with a key-name value that is equal to or greater than the value assigned to the indicated key.

- If key-name is a primary key, or is a secondary key that does not allow duplicates, the EQUAL TO option positions to the record in which that key field value is the same as the value assigned to key-name.

- If key-name is a secondary key that does allow duplicates, the EQUAL TO option will position to the first record with the indicated key value.

- START does not lock the record to which it positions.

START and Locked Records: If you attempt a START operation on a record that is already locked by another user, here's what will happen: a file-status code of 90 will be returned, you will (unintentionally) unlock the record for the other user, and your file position will remain unchanged. To make matters more interesting, the person who originally locked the record will subsequently get a 91 error when MIDAS attempts to unlock this already unlocked record.

In spite of all this, the original current record position remains unchanged File position is thus unaffected by a START that encounters a locked record. If another START is attempted, you'll get the record you want unless the other user (for whom you unlocked the record) beats you to it.

Out of Range Keys: If you specify a key value that causes START to position the file pointer to the end of the file, for example, START KEY IS GREATER THAN AGE where the AGE field has been initialized to its largest value in the file, the "end-of-file" condition will be raised by a subsequent READ. This makes sense because it is not an error to position the file pointer to end-of-file, but it is not possible to read beyond the last file record.

Examples: Generally, to process a file sequentially via some index, first set the file pointer to the beginning of that index this way:

```
MOVE LOW-VALUES TO key-name.
START filename KEY IS NOT LESS THAN key-name
INVALID KEY GO TO KEY-ERR.

                or

MOVE SPACES TO key-name
START filename KEY IS NOT LESS THAN key-name
INVALID KEY GO TO KEY-ERR.
```

On page 7-19, the phrase KEY IS should be inserted after PHONE-FILE in the example at the top of the page.

To set file position with a particular key value, simply move that value to the proper key field, as in:

```
MOVE '617' TO AREA-CODE.
START PHONE-FILE, NOT LESS THAN AREA-CODE INVALID KEY GO TO ERRORS.
                  ↳ KEY IS
```

Positioning on Partial Keys: can be done in SEQUENTIAL or DYNAMIC modes only, using the MOVE and START statements. The GREATER THAN and NOT LESS THAN options enable the use of partial key values in positioning the file pointer as long as the key value is correctly and fully initialized before the START statement is executed. This applies to both primary and secondary keys. For example, using the CUSTOMER file, if you wanted to find all the records whose CUST-NAME fields begin with the letter F and above, you might initialize the file position as shown in this program excerpt:

```
PROCEDURE DIVISION.
FIRST-PROC.
    OPEN INPUT CUSTOMER.
    MOVE 'F' TO CUST-NAME.
    START CUSTOMER KEY IS NOT LESS THAN CUST-NAME
    INVALID KEY GO TO KEY-ERR.
```

READING A FILE

File reads can be divided into two basic categories: sequential and keyed.

Sequential reads simply mean reading one record after the other in primary key or secondary key order depending on which index the file is positioned on. In this type of read, the user doesn't supply a key value except to tell MIDAS at which point in the index to start reading sequentially.

Keyed reads are also called "random" reads because it is possible to jump anywhere from the current file position by specifying a new key value to search for.

Access Modes

The three types of access modes possible in COBOL -- DYNAMIC, RANDOM and SEQUENTIAL -- were mentioned earlier. Each access mode permits only certain operations to be performed on a file. Keyed reads are the only type of read possible in RANDOM access mode, while sequential reads are the only type permitted in SEQUENTIAL access mode. DYNAMIC access mode allows the user to switch from one type of read to another, that is, you can do a keyed read to get to a certain spot in an index and then do sequential reads from there to retrieve the records which logically follow it.

Reminder: You must have the file open for INPUT or I-O in order to read it!

Sequential Reads

Sequential reads work by positiong the file to the next logical record in after the current record, making it current. This record is then read and returned to the user. This implies the need for a current record as a reference point. The current record is established by a MOVE and START or by a previous READ operation. Sequential reads are legal in SEQUENTIAL and DYNAMIC access modes, but not in RANDOM mode.

In SEQUENTIAL Access Mode: You can read the file sequentially by primary or secondary key. Records cannot be read randomly.

The format of a sequential READ statement in SEQUENTIAL access mode is:

```
READ filename [INTO read-var]
[AT END imperative-statement].
```

The optional INTO clause reads the record retrieved by the READ operation into read-var. If omitted, the record value is returned in the buffer associated with the file in the FD. The AT END clause must be included in each READ statement unless an applicable USE AFTER procedure has been specified for this file under the DECLARATIVES. The NEXT RECORD clause is implied but not stated in this format because every READ operation in SEQUENTIAL access mode automatically performs a position to the next record in the file before the READ is performed. Records are not locked when read if the file is opened for INPUT only; however, they are locked if the file is opened for I-O. Furthermore, the current record remains locked until another I/O operation is performed, yielding a new current record.

The following excerpt from a COBOL program written to access the CUSTOMER file shows the PROCEDURE DIVISION which reads the file sequentially, in primary key order.

```
PROCEDURE DIVISION.
FIRST-PROC.
    OPEN INPUT CUSTOMER.
    MOVE LOW-VALUES TO CUST-ID OF CUST-FILE-RECORD.
    START CUSTOMER KEY IS NOT LESS THAN CUST-ID
    INVALID KEY GO TO END-FILE.
READ-LOOP.
    READ CUSTOMER NEXT INTO READ-REC AT END GO TO END-FILE.
    DISPLAY READ-REC.
    GO TO READ-LOOP.
END-FILE.
     CLOSE CUSTOMER.
     IF STATUS-CODE =ØØ DISPLAY 'SUCCESSFUL COMPLETION.'
     ELSE
     DISPLAY 'STATUS CODE  IS:'  STATUS-CODE.
     STOP RUN.
```

In DYNAMIC Access Mode: You can read a file sequentially by primary or secondary key simply by using the NEXT clause. The key on which the READ is done could be established by a START or by a keyed read (see Keyed Reads, below). Once the file position is established (relative to a primary or secondary index), you can read the file sequentially, by index entry order, with this form of the READ statement:

```
READ filename NEXT RECORD [INTO read-var]
[AT END imperative-statement].
```

The AT END clause is used to trap end-of-file conditions, and must be specified if there isn't an applicable USE AFTER procedure under the DECLARATIVES. For example, if the CUSTOMER file is opened for I-O in DYNAMIC access mode, records can be read sequentially from some point in a primary or secondary index by just using START, then a READ NEXT statement:

```
PROCEDURE DIVISION.
FIRST-PROC.
    OPEN INPUT CUSTOMER.
    MOVE LOW-VALUES TO CUST-ID.
    START CUSTOMER KEY IS NOT LESS THAN CUST-ID
    INVALID KEY GO TO KEY-ERR.
READ-NEXT.
    READ CUSTOMER NEXT RECORD INTO READ-REC
      AT END GO TO KEY-ERR.
    DISPLAY 'READ NEXT AFTER START RETURNS FIRST LOG. RECORD'.
    DISPLAY READ-REC.
READ-CUR.
    DISPLAY 'READ WITH "KEY IS" RETURNS CURRENT RECORD'.
    READ CUSTOMER INTO READ-REC KEY IS CUST-ID
    INVALID KEY GO TO KEY-ERR.
    DISPLAY READ-REC.
READ-RANDOM.
    MOVE 'FLORA PORTRAITS   ' TO CUST-NAME.
    READ CUSTOMER INTO READ-REC KEY IS CUST-NAME INVALID KEY
    GO TO KEY-ERR.
    DISPLAY 'RANDOM READ ON SECONDARY KEY:' READ-REC.
    DISPLAY 'NOW READ SEQUENTIALLY ON THIS SECONDARY INDEX'.
READ-LOOP.
     READ CUSTOMER NEXT RECORD INTO READ-REC AT END
        GO TO FINIS.
     DISPLAY READ-REC.
    GO TO READ-LOOP.
KEY-ERR.
     DISPLAY 'STATUS CODE  IS:'  STATUS-CODE.
     GO TO CLOSE-FILES.
FINIS.
     IF STATUS-CODE = ØØ DISPLAY 'NO ERRORS'
      ELSE DISPLAY 'STATUS CODE IS:' STATUS-CODE.
CLOSE-FILES.
     CLOSE CUSTOMER.
     STOP RUN.
```

This program segment would read the entire file by secondary key (CUST-NAME) and would handle the end-of-file condition by transferring control to the FINIS label.

Keyed Reads

Keyed (random) reads are performed simply by specifying the key value on which a search should be conducted. Keyed reads are legal in RANDOM and DYNAMIC access modes and work the same way in each mode: move the key value into the proper key field, then use this form of the READ to position to and retrieve the desired record:

```
READ filename [RECORD] [INTO read-var]
[KEY IS key-name]
[INVALID KEY imperative-stmt].
```

For example, to retrieve a record for the CUSTOMER file with a CUST-ID value of '2194G', the program logic might be:

```
PROCEDURE DIVISION.
FIRST-PROC.
    OPEN INPUT CUSTOMER.
    MOVE '2194G' TO CUST-ID OF CUST-FILE-RECORD.
READ-RANDOM.
    READ CUSTOMER INTO READ-REC KEY IS CUST-ID
    INVALID KEY GO TO KEY-ERR.
    DISPLAY READ-REC.
```

A keyed read does not require, and in fact eliminates the need for, a START operation. In fact, STARTs are illegal in RANDOM access mode, which only allows keyed reads anyway. In RANDOM access mode, any READ done without the KEY IS clause automatically returns the current record, that is, the record to which the file pointer is pointing at the time the READ operation is encountered. It is not legal to MOVE LOW-VALUES to a key field prior to a keyed READ in RANDOM mode because there is no key value that matches LOW-VALUE (octal 200) in a MIDAS index subfile.

Changing Search Indexes

The "KEY IS" clause provides an easy method of switching from one index subfile to another without using a START. Simply put the key value you want to search into the proper key-name variable, then use that key-name in the KEY IS clause. This will establish key-name as the new "key of reference" and will automatically put you into the corresponding index subfile.

Reading Duplicates

For secondary keys that allow duplicates, it is possible to retrieve all the records with the same secondary key value in DYNAMIC access mode only. Follow these steps:

1. MOVE the desired secondary key value into the appropriate secondary key:

   MOVE 'sec-val' TO sec-key-name

2. Position the file with a START to the first record with this key value:

   START filename NOT LESS THAN sec-key-name INVALID KEY imperative-statement.

3. In a loop, use a READ NEXT statement with the AT END option to trap the end-of-file condition (status code 1Ø) which happens when there are no more entries in the index to search.

4. Compare the value just read with the value sought to verify that it is indeed a valid duplicate.

For example: Using the CUSTOMER file, suppose you want to read all the records which have the value WRCA in their LOCATION-CODE field (a secondary key). Position the file by secondary key to the first WRCA entry in secondary index subfile Ø2, then READ NEXT in a loop until no more duplicates are found. The following is the PROCEDURE DIVISION of a program that accesses duplicate values in the CUSTOMER file:

```
PROCEDURE DIVISION.
FIRST-DUP.
    OPEN INPUT CUSTOMER.
    MOVE 'WRCA' TO LOCATION-CODE.
     START CUSTOMER KEY IS NOT LESS THAN LOCATION-CODE
     INVALID KEY GO TO KEY-ERR.
READ-LOOP.
    READ CUSTOMER  NEXT RECORD INTO READ-REC
    AT END GO TO KEY-ERR.
    IF LOCATION-CODE NOT EQUAL 'WRCA' GO TO FINIS.
PRINT-REC.
    DISPLAY READ-REC.
    GO TO READ-LOOP.
KEY-ERR.
     IF STATUS-CODE = 1Ø  DISPLAY 'END OF FILE'
       ELSE
     DISPLAY 'STATUS CODE  IS:'  STATUS-CODE.
FINIS.
     CLOSE CUSTOMER.
     IF STATUS-CODE = ØØ DISPLAY 'ALL DONE'.
     STOP RUN.
```

When run, the program produces the following output:

```
9411PSTUDIO WEST              WRCA
9402AARTISTRY UNLTD.          WRCA
2334PSEACOAST STRIPPERS       WRCA
END OF FILE
ALL DONE
OK,
```

More on Partial Key Access

Partial key access is possible only by using the MOVE and START statements -- partial values cannot be used in READ operations. However, the MOVE and START statements provide a good method of searching for values less than or greater than a particular value. The value itself may represent a full or partial key value. By partial key value, we mean a prefix of a full key value. For example, if a key value is "FLORA", legal prefixes would include: "F", "FL", "FLO", and so forth. The use of partial key values in positioning the file was introduced earlier, and this example shows how this concept can be applied in a typical situation:

```
PROCEDURE DIVISION.
FIRST-PROC.
    OPEN INPUT CUSTOMER.
    MOVE 'F' TO CUST-NAME.
    START CUSTOMER KEY IS NOT LESS THAN CUST-NAME
    INVALID KEY GO TO KEY-ERR.
READ-IT.
    READ CUSTOMER NEXT RECORD INTO READ-REC AT END
    GO TO FINIS.
    DISPLAY 'FIRST RECORD WITH NAME BEG. WITH F:'
    DISPLAY READ-REC.
    DISPLAY 'READ REST OF RECORDS BEG. WITH F ON UP:'.
READ-LOOP.
    READ CUSTOMER NEXT RECORD INTO READ-REC AT END
    GO TO FINIS.
    DISPLAY READ-REC.
    GO TO READ-LOOP.
```

When run, the following is printed at the terminal:

```
FIRST RECORD WITH NAME BEG. WITH F:
1002PFLORA PORTRAITS          NENY
READ REST OF RECORDS BEG. WITH F ON UP:
4056SMARK-BURTON              NEMA
2334PSEACOAST STRIPPERS       WRCA
2194GSPECTROGRAPHICS          NWOR
9411PSTUDIO WEST              WRCA
```

ADDING RECORDS

Records can be added to a MIDAS file when it is opened for OUTPUT or I-O. However, a file opened for SEQUENTIAL access can only be written to if it's opened for OUTPUT. The WRITE statement takes information supplied by the user and adds it to the MIDAS file. Regardless of the order in which the records are presented, MIDAS inserts all primary key entries into the primary index subfile in ascending key sequence (low values first); however, the data records are always added to the bottom of the data subfile. Secondary key entries, like primary key entries, are added to secondary index subfiles in sorted order. Duplicates are handled a bit differently; when MIDAS first attempts to add a duplicate entry (for a secondary index that allows duplicates) it sets a flag in the original entry to indicate that there's more than one occurrence of this particular entry value in the index subfile. Duplicates are added sequentially thereafter, following the last matching key.

The WRITE Statement

A unique key value must be supplied for the primary key of each record added to an INDEXED SEQUENTIAL file. In other words, put a new value in the RECORD KEY (primary key) field before each WRITE statement is executed. To add secondary keys to their respective indexes, put the appropriate values in the secondary key fields prior to the execution of the WRITE statement. The WRITE statement format is:

```
WRITE record-name  FROM from-area
[INVALID KEY imperative-statement].
```

The important things about this statement are:

- In fixed-length files, if the data in from-area is not the same length as the file record, it is truncated or blank-filled.
- Make sure from-area and record-name do not reference the same memory location.
- A unique value for the primary key must be supplied prior to the execution of each WRITE.
- The INVALID KEY clause can be used to trap duplicate primary or secondary key errors and is required unless a USE AFTER procedure is specified for this file in the DECLARATIVES.

In files open for SEQUENTIAL access, the data values to be added should be supplied in sorted order, by primary key value. Because the file pointer cannot be randomly positioned in SEQUENTIAL access, the entries cannot be added randomly either. However, it's okay in the other access modes to give unsorted input to the program, although it just makes good sense (and better performance too) to sort it wherever possible.

WRITE Example: This program shows how to add entries to a file interactively by prompting the user for input, then inserting the entry into the file. This is a case where you can't always be sure of the order of input entries, so the file is opened for DYNAMIC access mode:

```
ID DIVISION.
PROGRAM-ID. ADD-PROG.
AUTHOR. LJD.
INSTALLATION. TPUBS.
DATE-WRITTEN. Ø8/28/8Ø.
DATE-COMPILED. Ø8/28/8Ø.
SECURITY. NONE.
REMARKS. PROGRAM TO TEST CUSTOMER FILE ADDS.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE COMPUTER. PRIME.
OBJECT COMPUTER. PRIME.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT CUSTOMER ASSIGN TO PFMS
        ORGANIZATION IS INDEXED
        ACCESS MODE IS DYNAMIC
        RECORD KEY IS CUST-ID
        ALTERNATE RECORD KEY IS CUST-NAME WITH DUPLICATES
        ALTERNATE RECORD KEY IS LOCATION-CODE WITH DUPLICATES
        FILE STATUS IS STATUS-CODE.
DATA DIVISION.
FILE SECTION.
FD  CUSTOMER
    LABEL RECORDS ARE STANDARD
    VALUE OF FILE-ID IS "CUSTOMER"
    DATA RECORD IS CUST-FILE-RECORD.
Ø1  CUST-FILE-RECORD.
    Ø2  CUST-ID PIC X(5).
    Ø2  CUST-NAME  PIC X(25).
    Ø2  LOCATION-CODE  PIC X(4).
    Ø2  FILLER PIC X(35).
WORKING-STORAGE SECTION.
Ø1   STATUS-CODE PIC 99.
Ø1  READ-REC PIC X(35).
PROCEDURE DIVISION.
FIRST-PROC.
    OPEN I-O CUSTOMER.
GET-DATA.
    DISPLAY 'ENTER CUSTOMER ID -- PIC X(5) OR ENTER XX TO QUIT'.
    ACCEPT CUST-ID.
```

```
           IF CUST-ID = 'XX' GO TO FINIS.
       CHECK-DUPS.
           READ CUSTOMER KEY IS CUST-ID
           INVALID KEY GO TO CHECK.
       GET-REST.
            DISPLAY 'ENTER CUST-NAME -- 25 CHARS'.
            ACCEPT CUST-NAME.
            DISPLAY 'ENTER LOCATION CODE -- 4 CHARS (REGION-STATE)'.
            ACCEPT LOCATION-CODE.
            WRITE CUST-FILE-RECORD INVALID KEY
              GO TO CHECK.
            GO TO GET-DATA.
       CHECK.
            IF STATUS-CODE = '23' GO TO GET-REST
             ELSE DISPLAY ' STATUS-CODE IS:' STATUS-CODE.
            GO TO GET-DATA.
       FINIS.
            IF STATUS-CODE = ØØ DISPLAY 'ALL DONE'
             ELSE
            DISPLAY 'STATUS CODE  IS:'  STATUS-CODE.
            CLOSE CUSTOMER.
            STOP RUN.
```

When run, the program does the following (user input is underlined):

```
ENTER CUSTOMER ID - - PIC X(5)  OR ENTER XX TO QUIT
8888T
ENTER CUST-NAME -- 25 CHARS
MARVIN'S TYPESETTING
ENTER LOCATION CODE -- 4 CHARS (REGION-STATE)
SWAZ
ENTER CUSTOMER ID - - PIC X(5)  OR ENTER XX TO QUIT
XX
ALL DONE
OK,
```

To verify that the new record was added and was inserted in the proper order, run the program shown earlier that reads the file sequentially by primary key:

```
OK, seq #seqrd
ENTER FILE ASSIGNMENTS:
> /
FILE ASSIGNMENTS COMPLETE.
Ø816SMorrow Paper Mills       NENH
1ØØ2PFLORA PORTRAITS          NENY
2194GSPECTROGRAPHICS          NWOR
2334PSEACOAST STRIPPERS       WRCA
4056SMARK-BURTON              NEMA
8888TMARVIN'S TYPESETTING     SSWA
9402AARTISTRY UNLTD.          WRCA
9411PSTUDIO WEST              WRCA
SUCCESSFUL COMPLETION.
OK,
```

UPDATING RECORDS (REWRITE)

A record update in COBOL consists of rewriting the entire record. Any field can be changed, with the exception of the primary key field. Because a record must be locked in order to be updated, only the current record can be updated. In SEQUENTIAL access mode, you must first READ the record to indicate which one is to be rewritten. In RANDOM mode you must position to the record with a keyed read; in DYNAMIC mode, either of these methods will do. If the record to be updated is not read prior to a REWRITE, a status code of 91 will be returned. This condition code indicates an unlocked record; a record must be locked in order to be updated. In addition, the file must be open for I-O: this applies to all access modes.

The REWRITE Format

The REWRITE statement format is the same for all access modes The INVALID KEY clause must be included in all REWRITE statements if there is no USE AFTER procedure specified for this file under the DECLARATIVES.

```
REWRITE record-name [FROM from-area]
[INVALID KEY imperative-statement].
```

If the FROM option is used, the RECORD KEY value must be the same as the key used in the previous READ. This option allows the new record to be written from another file or data area. The data in this "FROM area" is moved to the record-name buffer before it is written to the file. Without the FROM option, you directly modify the buffer (record-name) which contains the just-read data and then write it back to the file.

```
PROCEDURE DIVISION.
FIRST-PROC.
    OPEN I-O CUSTOMER.
    MOVE '2334P' TO CUST-ID.
READ-THIS.
    READ CUSTOMER INTO READ-REC KEY IS CUST-ID
    INVALID KEY GO TO KEY-ERR.
    DISPLAY 'CURRENT RECORD IS:' READ-REC.
 CHANGE-VAL.
    MOVE '2334PSEACOAST FINISHERS          WRCA' TO NEW-RECORD.
 UPDATE.
      REWRITE CUST-FILE-RECORD FROM NEW-RECORD
      INVALID KEY GO TO KEY-ERR.
      READ CUSTOMER INTO READ-REC INVALID KEY
       GO TO KEY-ERR.
      DISPLAY 'UPDATED RECORD IS:' READ-REC.
 ALT-UPDATE.
     MOVE  NEW-RECORD TO CUST-FILE-RECORD.
```

```
        REWRITE CUST-FILE-RECORD INVALID KEY GO TO KEY-ERR.
        GO TO FINIS.
    KEY-ERR.
        DISPLAY 'STATUS CODE  IS:'  STATUS-CODE.
    FINIS.
```

When run, the program produces the following output:

```
CURRENT RECORD IS:2334PSEACOAST STRIPPERS       WRCA
UPDATED RECORD IS:2334PSEACOAST FINISHERS       WRCA
ALL DONE
OK,
```

DELETING RECORDS

The DELETE statement in COBOL removes the data record and its primary index entry and marks all the corresponding secondary index entries for deletion. The space occupied by these secondary index entries is not reclaimed until the MPACK utility is run on this file. Remember that the file must be opened for I-O in order to delete entries from it.

The DELETE Statement Format

The DELETE format is:

```
DELETE filename [INVALID KEY imperative-stmt].
```

filename is the name assigned to the MIDAS file in the SELECT clause and FD clause. The INVALID KEY clause must be included in the DELETE statement when the file is opened for RANDOM or DYNAMIC access and there is no USE AFTER procdure specified for this file. Do not include the INVALID KEY clause in DELETE statements used on files opened for SEQUENTIAL access.

The important things to remember are:

- In SEQUENTIAL access mode, the record must first be read in order to be deleted. This is because a DELETE operation in SEQUENTIAL access mode does not perform a position operation: that's what the READ is for.

- In SEQUENTIAL access mode, the value in the RECORD KEY (the primary key) should not be changed between the READ and the DELETE statements. This is because DELETE can only operate on the current record and uses the primary key value used in the READ to check that it's the same one as the current record's primary key value.

- In DYNAMIC and RANDOM access modes, if the record for which a prior key value has been supplied cannot be found, the INVALID KEY clause will be activated and a file status code of 23 will be returned.

- A DELETE operation leaves the current record undefined, but a READ NEXT or a keyed read operation immediately after a DELETE will be successful, assuming that you didn't delete the last record in the file.

- You can't perform two DELETEs in a row in SEQUENTIAL access mode, but you can in RANDOM or DYNAMIC modes, as long as you supply a new primary key value.

How to Remove Records

To delete records in files opened for RANDOM or DYNAMIC access, MOVE the primary key value of the record to be deleted to the RECORD KEY. The DELETE statement then positions to this record and deletes it. In RANDOM and DYNAMIC access modes, DELETE automatically locks the record upon positioning to that record. This makes that record the current record. In SEQUENTIAL access mode, the record must be read prior to a DELETE to establish the current record position because DELETE does not perform a position operation in SEQUENTIAL access mode.

Note

If you attempt to read a record that has been deleted from a file, you'll get a 23 error (record not found). However, a subsequent READ NEXT will return the next record (in key order) that happens to follow the current record.

Examples

The first of these two examples shows the PROCEDURE DIVISION of a program that accesses the CUSTOMER file when opened for DYNAMIC access. The second program shows how to delete a record from the CUSTOMER file when it is opened for SEQUENTIAL access.

Program Excerpt 1:

```
       PROCEDURE DIVISION.
       FIRST-PROC.
           OPEN I-O CUSTOMER.
           MOVE '0816S' TO CUST-ID.
           DISPLAY 'DELETE RECORD WITH PRIMARY KEY 0816S'.
        DEL-REC.
             DELETE CUSTOMER.
             READ CUSTOMER NEXT RECORD INTO READ-REC AT END
               GO TO KEY-ERR.
            DISPLAY 'NEXT RECORD IS:' READ-REC.
            GO TO FINIS.
       KEY-ERR.
            DISPLAY 'STATUS CODE  IS:'  STATUS-CODE.
       FINIS.
            IF STATUS-CODE = 00 DISPLAY 'ALL DONE'
            ELSE DISPLAY 'STATUS CODE IS:' STATUS-CODE.
            CLOSE CUSTOMER.
            STOP RUN.
OK,
```

When run, the program produces this output:

```
DELETE RECORD WITH PRIMARY KEY 0816S
NEXT RECORD IS:1002PFLORA PORTRAITS          NENY
ALL DONE
```

Program Excerpt 2:

```
       PROCEDURE DIVISION.
       FIRST-PROC.
           OPEN I-O CUSTOMER.
           MOVE '2334P' TO CUST-ID.
           START CUSTOMER KEY IS EQUAL TO CUST-ID INVALID KEY
           GO TO KEY-ERR.
       READ-THIS.
           READ CUSTOMER INTO READ-REC
            INVALID KEY GO TO KEY-ERR.
           DISPLAY READ-REC.
        DEL-REC.
             DELETE CUSTOMER INVALID KEY GO TO KEY-ERR.
             GO TO FINIS.
       KEY-ERR.
           •DISPLAY 'STATUS CODE  IS:'  STATUS-CODE.
       FINIS.
            IF STATUS-CODE = ØØ DISPLAY 'ALL DONE'
            ELSE DISPLAY 'STATUS CODE IS:' STATUS-CODE.
            CLOSE CUSTOMER.
            STOP RUN.
OK,
```

The program produces the following output:

```
2334PSEACOAST FINISHERS       WRCA
ALL DONE
```

SECTION 8

THE BASIC/VM INTERFACE

INTRODUCTION

BASIC/VM's interface to MIDAS consists of a special set of file handling statements that read, write, delete and update entries in a MIDAS file. The MIDAS access statements are similar in format to the standard file handling statements in BASIC/VM, but extensions to their syntax permit the more complex operations associated with MIDAS files. Because the interface acts as a "go-between", translating the user's demands into the MIDAS subroutine calls which do all the work, a MIDAS file can be processed as easily as any other file type supported by BASIC/VM.

Note

This section assumes that you've read and are familiar with the the concepts in Section 1 and 2. In addition, it may be worth clarifying that access to MIDAS files is supported only by the BASIC/VM compiler, and not by any of Prime's other BASIC software products.

Language Dependencies

Like all other language interfaces (except PL/I), a template must be created for a MIDAS file (with CREATK) before it can be accessed from BASIC/VM. In addition, the following rules apply to BASIC/VM's MIDAS interface:

- Up to 17 secondary indexes are allowed per file (duplicates are allowed).
- Only keyed-index MIDAS files are supported; direct access MIDAS files cannot be processed by BASIC/VM.
- Key fields are not required to be part of the data record, but it is strongly recommended that they be included.
- The secondary data feature is not supported.

Two very important things to remember are: first, that keys are referred to by number in BASIC/VM; therefore the primary key is KEYØ, the first secondary is KEYØ1, the second KEYØ2, etc.; secondly, that key numbers and index subfile numbers are one and the same. Any reference to a particular key is implicitly a reference to the index subfile in which those key values are stored.

Summary of Access Statements

The BASIC/VM statements needed to process a MIDAS file are briefly summarized in Table 8-1. Experienced BASIC/VM users will note the similarity between these statements and those used in handling other file types in BASIC/VM.

OPENING/CLOSING A MIDAS FILE

BASIC/VM uses the same methods of opening and closing a MIDAS file as it does any file -- the DEFINE FILE statement. The file type must be declared as MIDAS.

The DEFINE FILE Statement

The DEFINE FILE statement opens a MIDAS file and assigns it a file unit number to be used as an alias for the file for the remainder of the program. Up to 12 files can be opened at a time from a single BASIC program. Be sure to use the unit numbers 1-12 when opening a MIDAS file. The keyword MIDAS is required. The format of this statement is:

```
DEFINE [READ] FILE #unit = filename, MIDAS [,record-size]
```

The parameters used in the above format are explained below.

Parameter	Meaning
#unit	The user-assigned unit number (either a literal or numeric expression); the # sign is required as in: #2.
filename	The name of the MIDAS file; must be enclosed in quotes, or must be a legal BASIC string expression.
record-size (optional)	The length of the MIDAS data subfile record in words; if the MIDAS file has fixed-length records, this number, if specified, must match the data size indicated in the MIDAS file. (Use the PRINT option of CREATK to determine this. See Section 12.) No record-size is necessary if the file has variable-length records. (The default record-size is 60 words.)

The READ option opens the file for reading only; no records can be added to the file. The default access mode allows the full range of file operations to be performed on the MIDAS file (no restrictions).

Table 8-1. Summary of Access Statements.

Statement	Description
ADD	Adds a record (by primary key) to the MIDAS file; secondary keys may be added also (optional).
DEFINE FILE	Opens the designated MIDAS file on an available PRIMOS file unit and assigns a BASIC/VM file unit number to it for program reference.
POSITION	Positions the file pointer by key value to a particular record in the file and locks it.
READ	Finds, locks and returns a MIDAS file record, by primary or secondary key; other options allow duplicate retrieval as well as sequential record retrieval.
REMOVE	Deletes a record by primary key; also, can delete any secondary index entry.
REWIND	Positions the file pointer to the first entry in the specified index (defaults to primary index).
UPDATE	Rewrites the current record.

All of these statements are explained in detail in the remainder of this section.

For example, the first of these two statements opens a file with fixed-length records; the second opens a file with variable-length records:

```
DEFINE FILE #1 = 'CUSTOMER', MIDAS, 35
DEFINE FILE #1 = 'SAMPLE', MIDAS
```

CLOSE Statement

MIDAS files are closed just like any other BASIC/VM file:

```
CLOSE #unit
```

The unit number is the user-assigned BASIC/VM file unit on which the MIDAS file is opened. The # sign is required.

Error Handling

The BASIC/VM ON ERROR statement can trap MIDAS errors by directing program control to a statement which will be executed if an error occurs. ON ERROR traps only I/O errors on the particular unit on which the file was opened. A general error trap, as well as one for each file opened, may be in effect simultaneously. Following an error, the MIDAS code of this error can be obtained by using MIDASERR. This works much like ERR, the special error-code variable that returns BASIC/VM error codes. You can then look up the MIDAS error code in Appendix A to determine the nature of the problem. The ON ERROR format is:

```
ON ERROR [#unit] GOTO line-number
```

#unit is the user-assigned unit number on which the MIDAS file was opened. If #unit is not specified, all I/O errors occurring on every opened file unit will be trapped. line-number is the line number of the first statement of the error handler.

To print out the MIDAS error code, use the following PRINT statement:

```
PRINT MIDASERR
```

FILE POSITIONING

Almost all of the BASIC/VM file handling statements use and/or set the current file position. The current file position is considered that record in the file at which the file "pointer" is pointing. The file position is actually established according to an index subfile. This makes sense because the only way records can be retrieved from the data subfile is via pointers to the records in the index subfiles. Thus the "file pointer" actually points to a

specific entry in some index subfile, which contains a pointer to a record in the data subfile, making that record the current record.

File position, then, can be established in the primary index subfile or in one of the secondary index subfiles. This can be done with the REWIND, POSITION or READ statements, all of which generally establish a new file position based on a user-specified key value. If the user does not supply a key value or an index number, the file position is set by default using the primary index. The "current record" would then be set to the record referenced by the first entry in the primary index subfile. It may be helpful to think of the index subfile which is pointing to the current record as the "current index."

Other BASIC/VM MIDAS access statements use the current file position to perform some action, and some of them reset it to a new location after the operation is complete. All of this is important only when trying to visualize where you are in the file at any given time. The idea of the "current record" and current file position will become clearer as you familiarize yourself with the way these statements actually work.

Locking/Unlocking

There are no "lock/unlock" statements in BASIC/VM; therefore, in order to preserve the integrity of any record, the READ, UPDATE, POSITION and DELETE statements all lock a record before they perform their respective operations. By locking the record, BASIC/VM protects the record from accidental harm by another user or process. It is a good idea to include error traps in your programs for records that may already be locked by another user when you try to access them. Specifically, one would want to trap for a MIDAS error number 14. (See Appendix A of this book for a list of MIDAS error codes.)

The POSITION Statement

The POSITION statement moves the file pointer to any record in the MIDAS file, making it the current record. POSITION locks the record, leaving it locked until the file pointer is positioned to another record.

The format is:

```
                                    SEQ
POSITION #unit [{,KEY [key-number] = key-value}]
                                   SAME KEY
```

The parameters and keywords used in the format are explained below.

Parameter	Meaning
key-number	Key number (index subfile number), which may be a literal or numeric expression; if unspecified or zero, it is taken as the primary key.
key-value	Key value enclosed in quotes, or a legal string expression.
SEQ	Positions pointer to the next sequential record in the file according to the order of entries in the current index.
SAME KEY	The next record is made current if it has the same key value as the current record; used when most recent file position was established via a secondary key allowing duplicates.

If there is no record at the file position specified, an error will be flagged. Some examples of the various POSITION options are:

```
POSITION #1, SAME KEY
POSITION #4, SEQ
POSITION #2, KEY 3 = '478'
```

How POSITION Works: Records are positioned by primary or secondary key, as specified on the POSITION statement line. This establishes that key as the "current key of reference" and it establishes the index subfile in which that key's value is stored, as the current "index of reference." If no key number is specified, the primary key is assumed, and POSITION uses the primary index subfile as its index of reference. (The primary key is then the "key of reference.") Once an index of reference is established, the file can be processed sequentially without explicitly referring to a key number or index subfile. All "read" requests are interpreted relative to that index; that is, records are read in the order in which their key values appear in the index of reference. (More on this under READING RECORDS.)

The REWIND Statement

The REWIND statement positions to and makes current the record with the lowest value for the specified key. If no key number is specified, the primary key is assumed. Essentially, REWIND just sets the file pointer relative to the first entry in the indicated index subfile. (Remember that references to key numbers are really references to index subfile numbers.) The REWIND statement format is:

```
REWIND #unit [, KEY num-expr]
```

num-expr is the key (index subfile) number; it must be used with KEY option.

Examples: The first example sets the file position to the record referenced by the initial entry in the primary index. The second sets the position to the record referenced by the first entry in secondary index subfile Ø3.

```
REWIND #3
REWIND #2, KEY 3
```

ADDING RECORDS

The ADD statement adds a record to a MIDAS file; it does not change the current file position or the current record. Although only the primary key value is required in an ADD, one or more secondary key values may be added to the appropriate index subfiles with a single ADD. In fact, this practice is recommended because it avoids the possibility of having index entries that don't match the key values in the data record. Whenever keys are being kept in the data record, it's important to make sure the entries in the primary and secondary index subfiles are the same as the key entries stored in the data subfile record; it minimizes confusion. The format of the ADD statement is:

```
                          PRIMKEY
ADD #unit, new-record,    KEY[Ø-expr]   = keyØ-val [,keylist]
```

where keylist has the form:

```
KEY key-number = key-val ...
```

This expression can be repeated for each secondary key field in the record. The key values assigned here should match the key values in the data record (new-record), if you are storing keys in the data record. The parameters are described below.

Parameter	Explanation
new-record	Data record to be added; should be equal in length to the record size declared for the file, if the file has fixed-length records; record should be padded to correct length with blanks. If you want keys stored in the record, make sure new-record includes all the key values.
PRIMKEY KEY[Ø-expr]	Both of these keywords represent the primary key. Ø-expr is a literal or numeric expression that evaluates to zero.
keyØ-val	Represents the primary key value: may be a string expression or a literal.
keylist	An optional list of secondary key numbers and values.
key-number	A numeric expression indicating a secondary key number (index subfile number).
key-val	A string expression or literal containing a secondary key value.

Only the keys which are explicitly specified in the keylist are entered in the respective index subfiles. It is recommended that you add all index entries at the same time to avoid possible ambiguities. The example below shows an ADD statement that adds all the index entries along with the data subfile entry (also called the "data record").

ADD Example

This example shows a BASIC/VM program which adds records to the CUSTOMER file created in Section 2. This program reads data from a sequential disk file, called NAMES, pads each record with blanks until it's the proper length, then adds each record to the data subfile. At the same time, the primary and secondary key entries are placed in the proper index subfiles, because all the necessary key values were included in the keylist. This example also shows the output from the program when it is executed. Notice that the program is run from PRIMOS command level, using the BASICV command.

```
OK, slist add.basic
10 DEFINE FILE #1 = 'CUSTOMER', MIDAS, 35
20 DEFINE FILE #2 = 'NAMES', 17
30    DEF FNS$(X$,N)  ! PADS STRING X$ WITH SPACES
40      ! ADD SPACES UNTIL THE LENGTH EQUALS 70 CHARACTERS
50        X$ = X$ +' ' UNTIL LEN(X$) = N
60        FNS$ = X$  ! ASSIGN THE NEW PADDED STRING TO FUNCTION
70    FNEND ! END OF PAD FUNCTION
80 J = 7
```

```
90 N = 70 ! NUMBER OF CHARACTERS PER RECORD
100  FOR I = 1 TO J
110    READ #2, A$
120    PRINT 'RECORD VALUE IS:': A$
130    B$ = SUB(A$,1,5)
140    PRINT 'PRIMARY KEY IS:': B$
160    C$ = SUB(A$,6,30)
170    D$ =SUB(A$,31,34)
180    ! PAD STUFF HERE
190    A$ = FNS$(A$,N)
200    ADD #1, A$, PRIMKEY = B$, KEY1 = C$, KEY2 = D$
210 NEXT I
220 CLOSE #1
230 CLOSE #2
240 PRINT 'DONE'
250 END
OK, basicv add.basic
RECORD VALUE IS: 2194GSpectrographics          NWOR
PRIMARY KEY IS: 2194G
RECORD VALUE IS: 1002PFlora Portraits          NENY
PRIMARY KEY IS: 1002P
RECORD VALUE IS: 9411PStudio West              WRCA
PRIMARY KEY IS: 9411P
RECORD VALUE IS: 9402AArtistry Unltd.          WRCA
PRIMARY KEY IS: 9402A
RECORD VALUE IS: 0816SMorrow Paper Mills       NENH
PRIMARY KEY IS: 0816S
RECORD VALUE IS: 2334PSeacoast Strippers       WRCA
PRIMARY KEY IS: 2334P
RECORD VALUE IS: 4056SMark-Burton              NEMA
PRIMARY KEY IS: 4056S
DONE
OK,
```

A Note on Record Size: Please note carefully the method used in the above example to pad the data record to the specified record length. This is mandatory in fixed-length record files or you'll get an error message when you attempt to add the record. If your file has variable-length records, don't worry about this.

READING RECORDS

The BASIC/VM READ statement provides users with a complete range of options for getting information out of a MIDAS file. You can read a MIDAS file both randomly and sequentially, on any index, and you can alternate easily between the two types of reads. A READ operation always locks the record to which it positions, guarding against concurrency errors. When the READ is complete, the record remains locked until another operation is performed to change the file pointer location.

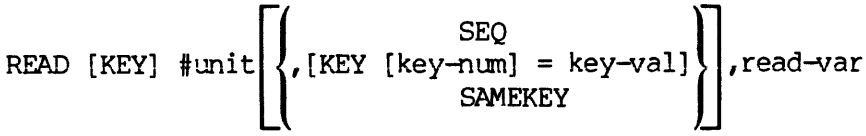
Overview of READ Options

With the SEQ option, the file can be processed sequentially on any index; this operation is equivalent to a "read next." The SAMEKEY option allows the retrieval of items with duplicate secondary key values. Partial key searches are permitted on any key; a special "READ KEY" option returns the full value of any key as stored in the appropriate index subfile.

An optionless READ returns the current record, which is the record to which the file pointer is currently positioned.

The READ Statement Format

The READ statement format allows you to read records from a MIDAS file sequentially, by duplicate keys and by primary or secondary key values. The format is:



The keywords and parameters used in this format are:

Parameter	Explanation
[KEY]	Optional; for reading the full value of any key as contained in an index subfile.
#unit	The user-assigned unit number on which the file is opened; the # sign is required.
KEY	This keyword is used with key-num and key-val to indicate which record should be read. If the KEY [key-num] = key-val clause is omitted, the current record is read. In this case, the current record must be established by a POSITION or REWIND statement.

key-num — A literal or numeric expression indicating which key (index subfile) should be used in this retrieval. If omitted, the default is Ø (the primary key).

key-val — The full or partial value (see below) of the key on which to conduct the search; if KEY [key-num] is used, a key-val must be supplied.

SEQ — Indicates that the next sequential record, as determined by current index of reference, should be read.

SAMEKEY — For reading duplicates; reads next record with the same key value as current record.

read-var — A string variable into which the retrieved record value is read.

READ Examples

The following examples illustrate some uses of the READ statement as applied to MIDAS files; all are taken from BASIC/VM programs designed to access the CUSTOMER file created in Section 2 of this book.

Sequential READs: The following program segment shows how to position the file by primary key and how to step through the file sequentially in primary key order:

```
OK, slist readseq.basic
10 DEFINE FILE #1 = 'CUSTOMER', MIDAS,35
20 READ #1, G$ ! READS FIRST ENTRY IN FILE
30 PRINT G$
40 ! READ SEQUENTIALLY THRU FILE ON PRIMARY INDEX
50 PRINT 'STEP THRU FILE ON PRIMARY INDEX'
60 PRINT
70 N= 6 ! NO. RECORDS REMAINING IN FILE
80  FOR I = 1 TO N
90    READ #1, SEQ, G$
100    PRINT G$
110  NEXT I
```

The output from this portion of the program would be:

```
OK, basicv readseq.basic
Ø816SMorrow Paper Mills        NENH
STEP THRU FILE ON PRIMARY INDEX

1ØØ2PFlora Portraits           NENY
2194GSpectrographics           NWOR
2334PSeacoast Strippers        WRCA
4Ø56SMark-Burton               NEMA
94Ø2AArtistry Unltd.           WRCA
9411PStudio West               WRCA
```

Similarly, if positioned by secondary key, which can be done with either a POSITION or READ statement the file can be read sequentially in secondary key order, as shown in the program excerpt below, which is the second half of the program shown above:

```
12Ø ! STEP THRU FILE ON A SECONDARY INDEX
13Ø PRINT
14Ø PRINT 'STEP THRU FILE ON SECONDARY INDEX Ø1'
15Ø PRINT
155 REM POSITION TO FIRST ENTRY IN INDEX SUBFILE Ø1
16Ø REWIND #1, KEYØ1
18Ø  FOR I = 1 TO 6
19Ø    READ #1,SEQ,X$ ! READ NEXT
2ØØ    PRINT X$
21Ø  NEXT I
22Ø CLOSE #1
23Ø END
```

The output from this portion of the program is:

```
STEP THRU FILE ON SECONDARY INDEX Ø1

1ØØ2PFlora Portraits           NENY
4Ø56SMark-Burton               NEMA
Ø816SMorrow Paper Mills        NENH
2334PSeacoast Strippers        WRCA
2194GSpectrographics           NWOR
9411PStudio West               WRCA
```

Random READs: To randomly locate a particular file value, use the "KEY [key-num] = key-val" option, as in:

```
READ #1, KEYØ2 = 'WRCA', G$
```

This READ returns the first record with the secondary key value 'WRCA'.

Finding Duplicates: To find all the records with the same secondary key value, use the SAMEKEY option, as in:

```
OK, slist dups.basic
10 DEFINE FILE #1 = 'CUSTOMER',MIDAS,35
20 ! THIS PROGRAM FINDS FIRST OCCURRENCE OF A CERTAIN
30 ! SECONDARY KEY VALUE AND THEN FINDS ALL THE DUPLICATES
40 READ #1,KEY02='WRCA', G$
50 PRINT 'FIRST RECORD WITH THIS VALUE:', G$
60 PRINT
70 ! NOW READ ALL DUPLICATES OF THIS KEY
80   ON ERROR GOTO 160
90 ! ERROR WILL OCCUR WHEN NO MORE KEYS ARE FOUND WITH THIS VAL
UE
100 PRINT 'RECORDS WITH DUPLICATE VALUES ARE:'
110 PRINT
120   FOR I = 1 TO 5
130    READ #1, SAMEKEY, S$
140    PRINT S$
150  NEXT I
160 ! ERROR HANDLER
170 PRINT 'BASIC ERROR IS:', ERR$(ERR)
180 CLOSE #1
190 END
OK, basicv dups.basic
FIRST RECORD WITH THIS VALUE:
9411PStudio West                WRCA

RECORDS WITH DUPLICATE VALUES ARE:

9402AArtistry Unltd.            WRCA
2334PSeacoast Strippers         WRCA
BASIC ERROR IS:       RECORD NOT FOUND
```

Obviously, this kind of program would only work when secondary keys allow duplicates, as explained in Section 2. The duplicate feature is turned on or off during template creation. The error handler is used to trap the error that will inevitably occur when we find no more duplicates for this key value.

Reading on Partial Key Values: Partial key values can be used in any READ statement, as in:

```
READ #1, KEY 1 = 'Flor', K$
```

The full value of this key is actually "Flora Portraits". Remember that only prefixes of a key value are allowed as partial key values. In other words, it would not be legal to do a search on "Portraits" in this case.

The READ KEY Option: The READ KEY option can be used to return the full value of any key. With it, you can find out what key you're currently positioned on, (that is, what index you're using

as the index of reference); alternatively, you can obtain the full key value of any key by specifying a partial value. For example:

```
OK, slist readkey.basic
10 DEFINE FILE #1 = 'CUSTOMER', MIDAS,35
20 ! READ RECORD WITH PARTIAL KEY
30 READ #1, KEY 1 = 'Flor', K$
40 PRINT 'RECORD READ WITH PARTIAL KEY:', K$
50 ! RETURN FULL VALUE OF CURRENT KEY OF REFERENCE
60 READ KEY #1, H$  ! READS CURRENT KEY POS'D TO
70 PRINT 'CURRENT KEY VALUE IS:': H$
80 ! READ KEY CAN ALSO BE DONE WITH A KEY SEARCH CLAUSE:
90 READ KEY #1, KEY2 = 'NE', K$
100 ! FIND RECORD WITH PARTIAL KEY VALUE
110 ! THEN RETURN FULL KEY VALUE
120 PRINT ! SPACE
130 PRINT 'FULL KEY :': K$ ! KEY USED IN READING RECORD IS READ
140 CLOSE #1
150 END
OK, basicv readkey.basic
RECORD READ WITH PARTIAL KEY:
1002PFlora Portraits          NENY
CURRENT KEY VALUE IS:
Flora Portraits

FULL KEY :
NEMA
OK,
```

UPDATING RECORDS

The UPDATE statement replaces the current record with a new record value. UPDATE does not change any of the index subfile entries so don't attempt to change key values with UPDATE. The proper way to change key values is to delete the record and then add it back again with the new key values. The UPDATE format is:

```
UPDATE #unit, new-record
```

#unit is the user-assigned file unit on which the file is open. new-record is the new data record value; can be a string variable or a quoted literal.

Since an update operation is usually done to modify a certain record, the record should first be read to establish it as the current record, and to return the contents of that record to be modified. It is not necessary to do a READ to establish the current record position, as a REWIND or POSITION statement can be used to the same purpose. The user should note that UPDATE overwrites the existing record -- it does not delete and replace it. Take care to make the new record equal in length to the old one so that all the old data is completely overwritten.

For example:

```
OK, slist update.basic
10 DEFINE FILE #1 = 'CUSTOMER', MIDAS,35
20 ! FIND RECORD TO BE UPDATED
30 READ #1, KEY1='SEA', X$
40 PRINT 'ORIGINAL RECORD IS:', X$
50 PRINT
60 ! UPDATE THIS RECORD BY ADDING SOMETHING TO THE END
70 A$= 'Contact Marvin'
80 ! WRITE THE ORIGINAL RECORD BACK WITH THIS ADDITION
90 X$ = SUB(X$,1,35) ! TAKE JUST THE NON-BLANK PART
100 X$ = X$+A$ ! COMBINE THE TWO
110 ! NOW PAD TO CORRECT LENGTH
120 X$ = X$ +' ' UNTIL LEN(X$) = 70
130 UPDATE #1,X$
140 REWIND #1
150 READ #1, KEY='2334', X$
160 PRINT 'UPDATED RECORD:', X$
170 CLOSE #1
180 END
OK, basicv update.basic
ORIGINAL RECORD IS:
2334PSeacoast Strippers       WRCA

UPDATED RECORD:
2334PSeacoast Strippers       WRCA Contact Marvin
```

In files with fixed-length records, be sure to pad the record to the correct length or you'll get a record-size error and the update will not occur.

DELETING RECORDS

The REMOVE statement selectively deletes index entries for a particular data record. If the primary key is specified, then the associated data record, as well as the primary index entries, will be deleted. In this case, the secondary key entries will not be deleted until they are used to reference the now deleted data record, or until MPACK is run on the file. (See Section 12.)

The REMOVE format is:

REMOVE #unit [,KEY [key-num] = key-val] [,KEY [key-num]=key-val]...

key-num is a numeric variable containing an optional key (index subfile) number. This is the key to be deleted. If a key number is not specified, the primary key is assumed. More than one key value can be deleted in a single REMOVE statement, as shown in the above format. key-val is a string expression containing a key value; along with key-num, it indicates which primary or secondary key entry is to be removed from an index subfile. To delete the current record, use the optionless form of REMOVE.

Example

The following example shows how specific secondary key values can be removed from an index, and how an entire record and its primary index entry can be deleted. In addition, it uses the MIDASERR feature to print out the MIDAS error code associated with the read error that will occur on an attempt to read a deleted record.

```
OK, slist delete.basic
10 DEFINE FILE #1 = 'CUSTOMER', MIDAS,35
20 ! REMOVE A SECONDARY INDEX ENTRY FROM THIS FILE
30 PRINT 'REMOVE THE SECONDARY KEY VALUE: Studio West'
40 REMOVE #1, KEY01='Studio West'
50 ! BUT THE RECORD VALUE REMAINS UNCHANGED
60 PRINT
70 READ #1, KEY = '9411P', X$
80 PRINT 'RECORD VALUE IS:':X$
90 REWIND #1, KEY01 ! POSITION TO TOP OF SECONDARY INDEX 01
100 ! READ FILE ON SECONDARY KEY
110 !
120   ON ERROR GOTO 210
130 PRINT
140 PRINT 'RECORDS READ BY SECONDARY KEY:'
150 PRINT
160   FOR I = 1 TO 6
170     READ #1,SEQ,N$
180     PRINT N$
190   NEXT I
200 ! DELETE THE RECORD
210 PRINT
220 PRINT 'NOTE: RECORD REFERENCED BY DELETED '
230 PRINT 'INDEX ENTRY IS NOT PRINTED'
240 PRINT
250 PRINT 'DELETE RECORD BY PRIMARY KEY'
260 REMOVE #1, KEY0='9411P'
270 REWIND #1
280 !
290   ON ERROR GOTO 340
300 PRINT
310 READ #1, KEY='9411', X$
```

```
320 PRINT X$
330 GOTO 360 ! IF NO ERROR
340 PRINT 'MIDAS ERROR CODE:': MIDASERR
350 PRINT 'BASIC ERROR IS:': ERR$(ERR)
360 CLOSE #1
370 END

OK, basicv delete.basic
REMOVE THE SECONDARY KEY VALUE: Studio West

RECORD VALUE IS:
9411PStudio West              WRCA

RECORDS READ BY SECONDARY KEY:

1002PFlora Portraits          NENY
4056SMark-Burton              NEMA
0816SMorrow Paper Mills       NENH
2334PSeacoast Strippers       WRCA
2194GSpectrographics          NWOR

NOTE: RECORD REFERENCED BY DELETED
INDEX ENTRY IS NOT PRINTED

DELETE RECORD BY PRIMARY KEY

MIDAS ERROR CODE: 7
BASIC ERROR IS: RECORD NOT FOUND

OK,
```

The MIDAS error code of 7 indicates an unsuccessful read resulting from a failure to find a record with the indicated key value. You can verify this by looking up MIDAS error 7 in Appendix A.

SECTION 9

THE PL/I INTERFACE

INTRODUCTION

This section documents the PL/I subset G interface to MIDAS files as implemented at this revision of MIDAS (17.6). The term PL/I as it is used in this section refers exclusively to Prime's Subset G version of the PL/I language unless otherwise indicated.

From PL/I's point of view, a MIDAS file is simply a RECORD KEYED SEQUENTIAL file that can be accessed by the standard PL/I READ, WRITE, REWRITE and DELETE statements. However, the MIDAS file must have an ASCII primary key: this is a PL/I requirement. In addition to supporting CREATK-defined files, PL/I can create its own MIDAS files. A PL/I-created file has an ASCII primary key of 32 characters in length and variable-length records. Further restrictions on the PL/I interface are discussed later.

In this section, the syntax and usage of PL/I statements are addressed in relation to MIDAS only; consult The PL/I Subset G Reference Guide for complete syntax information on these and other PL/I statements referenced here.

Language Limitations

The PL/I Subset G interface to MIDAS does not support the following MIDAS features:

- non-ASCII primary keys
- Secondary keys
- Direct access MIDAS files
- Secondary data

Because PL/I does not support secondary keys or non-ASCII primary keys you can't use PL/I to set up a MIDAS file template with these features. To create a MIDAS file with secondary keys, fixed-length records or a primary key of less than 32 characters, use CREATK instead. You can access such a file from a PL/I program as long as its primary key is a character string of 32 characters or less. Existing MIDAS files with secondary keys can still be accessed from a PL/I program, but you will not be able to access any secondary index subfiles from PL/I.

Note on Conversion

The restriction on primary key type applies only to the actual definition of the primary key during template creation. As long as the primary key is declared as an ASCII character string of 32 or fewer characters, the file can be accessed from PL/I using character or numeric key values. Numeric values will be converted to character strings in accordance with standard PL/I conversion rules.

Running a PL/I Program

You don't need to load any special libraries in order to run a PL/I program that accesses MIDAS files. Just follow this sample compile and load sequence. User input is underlined to distinguish it from system output.

```
OK, pl1g program
0000 ERRORS (PL1G-REV 17.6)

OK, seg
[SEG rev 17.6]
# lo #program
$ lo b_program
$ li pl1lib
$ li
LOAD COMPLETE
$ sa
$ q

OK, seg #program
```

Substitute the name of your program for the program argument in the above example.

OPENING/CREATING A MIDAS FILE

There are two ways to create a MIDAS file for use with PL/I: you can create the file with CREATK, and then simply open it for reading and/or writing from a PL/I program; or, you can create a MIDAS file from PL/I using the standard file I/O statements. The next part of this section tells how to open an existing MIDAS file and how to create a new one with PL/I.

Creating a MIDAS File from PL/I

When creating a MIDAS file from a PL/I program the file must be explicitly declared with the KEYED SEQUENTIAL [RECORD] attributes. Their order doesn't matter. The RECORD attribute is implied by the

KEYED attribute and need not be specified. Follow this simple sequence in creating a MIDAS file from PL/I:

```
DECLARE filename FILE KEYED SEQUENTIAL;
OPEN FILE(filename) OUTPUT;
```

The FILE keyword in the DECLARE statement can appear before or after the actual filename. The DECLARE statement can be abbreviated to DCL. These statements tell MIDAS to open a MIDAS file with the default attributes mentioned above. (See The PL/I Subset G Reference Guide for details on OPEN and DECLARE syntax and attributes.) The MIDAS file will have variable-length records with a maximum length of 2048K words (4096 bytes) and a primary key of 32 characters in length.

Although the default primary key is defined as an ASCII character string, PL/I conversion rules permit you to then define the primary key from program level as a character or numeric item. Thus you can write file entries with character or numeric key values. Keep in mind that the data must be read out in the same format as it was written. You can't mix and match data types in WRITEs and READs.

Opening an Existing MIDAS File

To open an existing MIDAS file that was previously created with PL/I, use the OPEN statement with this format:

```
OPEN FILE(filename)  {INPUT }
                     {OUTPUT}
                     {UPDATE}
```

filename must be no longer than 8 characters in length, and may be specified before or after the FILE option. The file should first be declared as KEYED SEQUENTIAL. The INPUT, OUTPUT and UPDATE options indicate the access mode in which the file should be opened. The access mode controls the types of operations which can be performed on the file.

The INPUT option: allows READ operations only. It defines the access mode for this file as "read only".

The OUTPUT option: permits only write operations and is primarily used to open and create new files. It also allows additions to be made to an existing KEYED SEQUENTIAL file with the indicated name; however, if no such file exists, PL/I creates a new one. Neither read nor update operations cannot be performed on a file opened for OUTPUT.

The UPDATE option: permits READ, WRITE, UPDATE and DELETE operations on an existing file. Use UPDATE when making changes or additions to an existing file.

Combining DCL and OPEN: Alternatively, the INPUT, OUTPUT or UPDATE options can be included in the DECLARE statement where the KEYED SEQUENTIAL file is first declared, eliminating the need for an explicit OPEN statement:

```
DECLARE SAMPLE FILE KEYED SEQUENTIAL OUTPUT;

            or

DCL OTHER FILE KEYED SEQUENTIAL INPUT;
```

Remember that a file opened or declared as INPUT or UPDATE must already exist -- if it doesn't, an error is signalled.

Note

For information on opening non-PL/I-created files, (i.e., MIDAS files created directly with CREATK or KX$CRE), see Opening CREATK-defined Files at the end of this section.

FILE I/O CONCEPTS IN PL/I

The remainder of this section deals with these PL/I statements, which are all used to access MIDAS files from PL/I programs:

PL/I Statement	Function
READ [KEY]	Reads next sequential record in file (if KEY not specified) or reads record with the indicated KEY value.
WRITE	Adds a new record to the bottom of the file with a specified primary KEY value.
REWRITE [KEY]	Updates (rewrites) the indicated record (KEY specified) or the current record (no KEY).
DELETE [KEY]	Deletes the specified record (KEY specified) or the current record (no KEY).

The access mode for which the file is opened restricts the use of these statements.

The table below indicates which statements can be used in each access mode:

INPUT	OUTPUT	UPDATE
READ	WRITE	READ WRITE REWRITE DELETE

The topics addressed in the remainder of this section are:

- The current record in PL/I file I/O
- Writing to a file (new or existing)
- Data size/record length
- Reading a file (sequentially or by key)
- Updating file records
- Deleting file records
- Locked records
- Accessing CREATK-defined files

The Current Record in PL/I

In PL/I the current file position is handled automatically so you don't have to worry about a communications array to keep track of the current record (as in the FORTRAN interface). PL/I's READ statement advances the current file position pointer to the next sequential record in the file before performing the read operation. After a READ, the current record is always the one just read. The WRITE statement positions to the proper spot in the primary index subfile so it can insert the primary key value of the record being added in its proper place. Thus the current record after a WRITE is the record just written. When a DELETE is performed, the current record position is not moved, but is instead left undefined. That is, no record is "current" after a DELETE because DELETE always removes the current record. REWRITE does not update the current record position at all.

Initial "Current" Record: When a MIDAS file is opened for INPUT or UPDATE in PL/I, the current file pointer is set just prior to the first record in the file; a current record position is established by a READ (with or without KEY), or any other I/O operation permitted by the access mode. This establishes the current file position and initializes the MIDAS communications array which PL/I handles (transparently) for the user. (This means you don't have to worry about it!).

DELETE and the Current Record: After a DELETE operation, the current record is left undefined until the next READ or WRITE operation. A sequential READ (without the KEY option) will work unless you just deleted the last record in the file. A WRITE operation positions the file pointer to the place in the index where the key entry associated with the record to be written logically fits according to the collating sequence.

ADDING RECORDS

To add records to a new or existing file, the file must first be opened for OUTPUT or UPDATE. Records are then written to the MIDAS file with a PL/I WRITE statement:

```
WRITE FILE(filename) FROM(var) KEYFROM(keyvar);
```

The var argument contains the new record information. Its data type should be declared as CHARACTER; its size, which PL/I views as the record size, can be declared varying (VAR) if desired. (See Declaring Data Size below.)

The KEYFROM Option

The KEYFROM option specifies the primary key value for this new record; KEYFROM and a unique key value must be present in every WRITE statement performed on a MIDAS file. keyvar can be declared as a character string of 32 characters or less, or as a numeric field ,for example, fixed bin, fixed decimal, and so forth. If declared as a CHARACTER, it cannot be declared as VARYING. Furthermore, if the template was created with CREATK, the variable should match in size and type the primary key as defined during template creation.

Inside Story: PL/I always writes 32 characters per index entry, regardless of how many non-blank characters you specify in a WRITE statement and regardless of how you define the primary key in your program. It may be wise to declare the key variable as 32 characters even if you only have a 6 character key, for example. It's better to have PL/I add the remaining 26 characters as blanks than to have unpredictable "garbage" values stored in the index entry slot.

No Duplicates Allowed: The key value specified for keyvar must not already exist in the file or a KEY error will be reported, causing program execution to halt. Remember, primary keys must be unique in a MIDAS file -- you must supply a new value for the keyvar argument with every WRITE statement. (For more details on KEY errors, see Error Handling, below.)

The WRITE Operation

WRITE updates the current record position in order to add a new record to the file. It doesn't matter which record is current prior to a WRITE operation, because WRITE takes care of positioning the file to the proper index location. The current record after a WRITE is the one just written, so a "read next record" operation subsequent to a WRITE returns the record immediately following the one just added. However, the file must be opened for I/O or OUTPUT in order to add records to it. Each WRITE operation places key entries in their proper slots in the index the program-supplied primary key entry into its proper slot in the index and adds the corresponding data record to the bottom of the data subfile. Keys are added to the primary index subfile in ascending order by key value regardless of the fact that data subfile records are written to the bottom of the data subfile. This means that when reading sequentially through the file, you'll get all the records in the order you expect (based on primary index entry order) rather than in the order in which you added them — unless, of course, you added them in ascending key order in the first place!

Declaring Data Size

It is not possible to create a true fixed-length record MIDAS file from a PL/I program. However, you must declare a maximum size for the data variable from which each MIDAS file record will be written. This is the variable into which the user's program puts data record information so it can be written to the MIDAS file as a unit. By setting the size of this variable, you effectively limit the record size of the MIDAS file:

```
DECLARE SAMPLE FILE KEYED SEQUENTIAL;
DECLARE PKEY CHAR(32);
DECLARE DATAVAR CHAR(30);
PKEY = 'aaaa';
WRITE FILE(SAMPLE) FROM(DATAVAR) KEYFROM(PKEY);
```

In this example, datavar is set at 30 characters, indicating that the records written to the MIDAS file SAMPLE will be 30 characters in length. datavar could be declared as "CHAR(30) VAR" to eliminate blank padding. PKEY represents the primary key field for each file record and is set to the default length of 32 characters. It doesn't have to be 32 characters long -- this is the simply the maximum key size allowed by PL/I.

For Example: The OPENIT program, listed below, opens a new MIDAS file called SAMPLE and adds records to it. Primary key values are supplied in ASCII form because pkey is declared as char(32). If we wanted to, we could have declared the primary key as a fixed decimal number and then supplied the primary key values in numeric form.

```
openit:
        proc options(main);

/* This program creates a MIDAS file called Sample */

dcl sample file keyed sequential;                    /* MIDAS */
dcl pkey char(32);                                   /* primary key */

/* recvar contains the data to add to the file */

dcl recvar char(30) var;                             /* record size */
        open file(sample) output;                    /* for new file */

/* Values for pkey and recvar */

        pkey = '0001';
        recvar = 'first file record';
        write file(sample) from(recvar) keyfrom(pkey);
        pkey = '0002';
        recvar = 'second file record';
        write file(sample) from(recvar) keyfrom(pkey);
        pkey = '0003';
        recvar = 'third file record';
        write file(sample) from (recvar) keyfrom(pkey);
        close file(sample);
        end;
```

Storing Primary Keys in Record

To maintain data integrity and to allow for future file requirements, it might be a good idea to include the primary key in the data record proper. This can be done easily as shown in this example:

```
add:
        proc options(main);

/* this program adds a new record to the Sample file */

dcl sample file keyed sequential;
dcl pkey char(32);                                   /* primary key */
dcl recvar char(30) var;
        open file(sample) output;

/* output mode is ok for adding records only */

/* primary key is in data record */

        recvar = '0005fifth file record';
        pkey = substr(recvar, 1, 4);
        write file(sample) from(recvar) keyfrom(pkey);
        close file(sample);
        end;
```

When reading the file, remember to use "PUT EDIT" to print out just the part of the record you want. The key can also be stored at the end of the data record. Avoid making changes to the "key field" of the record during an UPDATE because the primary key entry in the index subfile cannot be changed.

READING A MIDAS FILE

There are three types of "file reads" that can be done on a MIDAS file from PL/I:

- Keyed read - a record is found based on a user-supplied key value
- Sequential read - (also called "non-keyed" read) records can be read from the file in primary key order; file position is established at some point in the primary index subfile, either by a keyed read or by default, which puts the file pointer at the beginning of the index subfile.
- Reading keys - the full key value of the primary key, as MIDAS stores it in the primary index, is returned (can only be done in a non-keyed read)

The READ Statement

The READ statement, with or without the KEY or KEYTO options, copies the contents of one file record into a previously defined variable. Reminder: a file must be opened for INPUT or UPDATE in order to be read. The READ statement format is:

```
READ FILE(filename) INTO(var) [KEY(keyvar)] [KEYTO(curkey)];
```

The KEY and KEYTO Options

The KEY (keyvar) option finds the record whose key matches the one specified by keyvar. KEY is used in keyed reads only (see Keyed Reads below); if omitted, the next sequential record is read. The keyvar argument should match the data type and size of the primary key as defined in the program that wrote entries to the file. For example, if you wrote records using a primary key declared as FIXED(4), you should read the file with the primary key declared as FIXED(4).

The KEYTO option copies the key value for the current record into the curkey argument. (The key value is read from the primary index subfile.) The curkey argument must always be declared as CHAR(32) VARYING, because of the way in which PL/I handles the MIDAS index subfile entries.

Note

The KEY and KEYTO options cannot appear together in the same READ statement. KEY is generally used for keyed reads when the key value of a record is known; KEYTO, on the other hand, is used during sequential reads when you want to determine the primary key value of the current record. KEYTO is especially useful where keys are not stored in the actual data file record.

The "Read" Variable

The record is read into the var variable, which must be declared according to the following rules:

- For PL/I-created files, var must match in size and type the WRITE argument which was used in writing this record. For example, the recvar argument, as used in:

  ```
  WRITE FILE(filename) FROM(recvar) KEYFROM(pkey);
  ```

  must match the var argument as used in the above format.

- For non-PL/I-created MIDAS files with fixed-length records, simply use the fixed record size value in declaring the INTO argument. See Reading Fixed-Length Records, below.

- For variable-length records in a file not created by PL/I, see Reading Variable-Length Records later in this section.

Keyed Reads

Keyed reads are performed by specifying a valid key value with the KEY option. "Valid" means that the value must occur in the primary index subfile of the MIDAS file being read. Key values that do not appear in the MIDAS file cause KEY errors and program halts. If a match is found, this record becomes the current record, and the contents of the record are placed in the specified read variable. For example:

```
DECLARE SAMPLE FILE KEYED SEQUENTIAL;
DECLARE PKEY CHAR(32);
DECLARE READVAR CHAR(30);
PKEY = 'aaaa';
READ FILE(SAMPLE) INTO(READVAR) KEY(PKEY);
PUT LIST(READVAR);
```

Sequential Reads

A MIDAS file can be read sequentially, in primary key order, by using READ without the KEY option. The current record pointer simply advances to the next record in the file every time a READ operation is

performed. An error occurs if the pointer is at the bottom of the file because there are no more records to be read. An example of a sequential read might be:

```
ON ENDFILE(SAMPLE) GOTO CLOSE_FILES;
DO WHILE('1'B); /* infinite loop */
READ FILE(SAMPLE) INTO(READVAR);
PUT SKIP LIST(READVAR);
END;
```

All the records in the file from the current record on will be read as the program loops; then the ENDFILE condition will be signalled and control sent to the part of the program labelled "CLOSE_FILES" where the file is closed.

Reading Key Values

The KEYTO option can be used to obtain the value of the primary key of the record currently being read:

```
DECLARE KVAR CHAR(32) VAR;
READ FILE(SAMPLE) INTO(READVAR) KEYTO(KVAR);
PUT SKIP LIST('RECORD:', READVAR);
PUT SKIP EDIT('KEY VALUE:', KVAR) (A, X(2), A(4));
```

The "PUT EDIT" statement is useful when you want PL/I to print only the first few characters of the primary index subfile entry. Otherwise, PL/I prints a 32-character version of the primary key which in this case consists of the four characters originally passed to MIDAS plus 28 blanks.

Sample Read Program: Below is a listing of a sample program which performs keyed and sequential reads on the file created by the OPENIT program listed earlier, along with the output from the program:

```
read:
        proc options(main);
dcl sample file keyed sequential;
dcl pkey char(32);                              /* primary key */
dcl readvar char(3Ø) var;
/* make it the same as recvar */
dcl kvar char(32) var;                          /* reads keys */

/* KVAR is used with KEYTO option and must be CHAR VARYING */

/* set up an on-unit to handle end of file  */

        on endfile(sample) begin;
            close file(sample);
            put skip list('End of file');
            stop;
            end;
        open file(sample) input;
```

```
/* read with a key */

/* Note: the first READ doesn't have to be a keyed one */

        pkey = '0001';
        read file(sample) into(readvar) key(pkey);
        put skip list(readvar);

/* now read sequentially (without key) */

        read file(sample) into(readvar);
        put skip list(readvar);

/* read with KEYTO option */

/* now read next record and return the key value with KEYTO
   option */

        read file(sample) into(readvar) keyto(kvar);
        put skip list(readvar);
        put skip edit('Key value:', kvar) (a, x(2), a(4));
        close file(sample);
        end;
/* now run the program */
OK, seg #read

first file record
second file record
third file record
Key value:  0003
OK,
```

UPDATING FILE RECORDS

Record updates in PL/I are performed with the REWRITE statement, which replaces either the current record (that is, the one just read) or the specified record with a new record value.

The REWRITE Statement

Records in an existing MIDAS file can be updated with the REWRITE statement:

```
REWRITE FILE(filename) FROM(datavar) [KEY(keyvar)];
```

datavar is the variable containing the data which will replace the record being updated. You can't update just part of a record -- you must rewrite the whole thing. Bear in mind that keys cannot be changed -- so if you've made an error while adding the original key field, delete the record and then WRITE it over the way you want it.

Sample Update Program: This program performs an update on the MIDAS file Sample. The output from the program is also included:

```
update:
        proc options(main);
dcl sample file keyed sequential;
dcl pkey char(32);                                  /* primary key */
dcl readvar char(30) var;
/* make it same as recvar */
dcl kvar char(32) var;                              /* reads keys */

/* KVAR is used with KEYTO option and must be CHAR VARYING */

/* set up an on-unit to handle end of file  */

        on endfile(sample) begin;
            close file(sample);
            put skip list('End of file');
            stop;
            end;
        open file(sample) update;
        pkey = '0002';
        read file(sample) into(readvar) key(pkey);
        put skip list(readvar);

/* update this record */

        readvar = 'New second record';

/* use the KEY option to be sure */

        rewrite file(sample) from(readvar) key(pkey);
        read file(sample) into(readvar) key(pkey);
        put skip list('New record:', readvar);
        close file(sample);
        end;
/* now run the program */
OK, seg #update

second file record
New record:   New second record
OK,
```

REWRITE's KEY Option

The KEY option is used when you want to specify exactly which record is to be updated. Without the KEY option, REWRITE updates the current record, which is the record just read by the last READ statement, or the one just written by the last WRITE statement. This means that a current record position must have already been established (by a READ, WRITE or another REWRITE, this one with the KEY option) prior to a REWRITE without the KEY option. In this case, the MIDAS communication

array is used by PL/I to determine where the current record is and which record should be updated. However, it is not necessary to perform either a READ or WRITE prior to REWRITE if the KEY option is used with REWRITE.

In any event, it's good practice to use the KEY option to avoid confusion about which record is being updated. If the key value indicated by keyvar does not exist in the file, a KEY error is triggered and the program aborts.

What REWRITE Does

REWRITE uses either the communications array value or the supplied key value to determine which record is "current". The current record is automatically locked during a REWRITE and kept locked until another I/O operation is performed. In other words, the current record position is not updated after the REWRITE operation is complete.

DELETING RECORDS

Records are deleted from a MIDAS file by primary key; the index subfile entry is marked for deletion and the corresponding primary index value is deleted.

The DELETE Statement

To delete a record from a MIDAS file, use the DELETE statement. DELETE may be used with a KEY option to indicate which record is to be deleted:

```
DELETE FILE(filename) [KEY(keyvar)];
```

If specified, keyvar must be a key value that occurs in the MIDAS file, otherwise a KEY error occurs. If no KEY is specified, the current record is deleted. (It is assumed that the previous READ or REWRITE statement established the current record position.) DELETE does not update the file pointer location -- thus, the current record is always left undefined.

For example, this program excerpt deletes a record from the Sample file:

```
delete:
        proc options(main);
dcl sample file keyed sequential;              /* existing file
*/
dcl pkey char(32);                             /* primary key */
dcl recvar char(30) var;
dcl readvar char(30) var;
dcl kvar char(32) var;                         /* reads keys */
```

On page 9-15, the second and third comment lines from the top of the page should be omitted. This program contains no on-units for the KEY condition (using an invalid key in a file access operation). This is why the error messages shown in the output occur. To help clarify this, the following sentence should be added before Reminders, on page 9-15:

> "The error conditions are raised because there is no on-unit to trap KEY errors: see ERROR HANDLING below."

```
/* KVAR is used with KEYTO option and must be CHAR VARYING */

/* Set up on-unit to handle file read errors */

/* set up an on-unit to handle end of file  */

        open file(sample) update;
/* UPDATE mode required for rewrites or deletes */
        pkey = '0002';
        read file(sample) into(readvar) key(pkey);
        put skip list(readvar);

/* delete this record */

        delete file(sample);

/* check to see if this record is gone */

/* if it is, a KEY error will be raised */

        read file(sample) into(readvar) key('0002');
        close file(sample);
        end;
/* now run the program */
OK, seg #delete

New second record
KEY(SAMPLE) raised in DELETE at 4001(3)/1211
(record not found in READ)

ERROR (file = SAMPLE) raised in DELETE at 4001(3)/1211
(no on-unit for KEY)
ER!
```

Reminders: A file must be opened for UPDATE in order to delete records from it. A READ or WRITE operation performed immediately after a DELETE works fine; however, a DELETE (with or without a KEY) or a REWRITE operation subsequent to a DELETE will signal an error.
THE ERROR CONDITIONS ARE RAISED BECAUSE THERE IS NO ON-UNIT TO TRAP KEY ERRORS: see ERROR HANDLING BELOW.

LOCKED RECORDS

Because the PL/I READ and REWRITE statements always lock the current record, there are no specific lock/unlock statements in PL/I. The lack of such statements can cause some problems if the user hits CTRL-P or BREAK immediately after a READ or REWRITE operation. Frequently, the current record will remain locked, since READ automatically sets and locks the current record even though it never updates the current file position. When this happens, run the MPACK utility to unlock this record. MPACK is documented in Section 12.

ACCESSING CREATK-DEFINED FILES

MIDAS files that are not created through PL/I can still be accessed from PL/I programs. However, keep in mind the restrictions on READ and WRITE argument size. It's easier to determine how the arguments should be declared if the file in question has fixed-length records.

Caution

It may be a good idea to avoid updating the same MIDAS file with more than one high-level language interface, as this practice can result in file anomalies.

Reading Fixed-Length Records

When opening an existing CREATK-defined MIDAS file with fixed-length records, determine the record size so you can use it in defining the READ or WRITE statement arguments. Use CREATK's PRINT function and enter "data" in response to the "INDEX NO?" prompt (See Section 12.)

The data size in words is displayed next to "ENTRY SIZE". (Since PL/I cannot create a MIDAS file with fixed-length records, the ENTRY SIZE is displayed as "USER-SUPPLIED" for PL/I-created MIDAS files.) For example, the CUSTOMER file, created in Section 2, has fixed-length records of 35 words. It can be opened and accessed as shown in this PL/I program excerpt:

```
DCL FILE CUSTOMERS KEYED SEQUENTIAL;
OPEN FILE CUSTOMERS INPUT; /* could open it for UPDATE also */
DCL READVAR CHAR(70);      /* variable into which record is read */
DCL PKEY CHAR(5);
PKEY = '0212G';
READ FILE(CUSTOMERS) INTO(READVAR) KEY(PKEY);
```

Since the record size is already known, the readvar variable is set at 70 characters. Similarly, the primary key, defined here as pkey, was declared as 5 characters when the file was created with CREATK.

Sample Program: This program, from which the previous excerpt was taken, opens and reads the sample MIDAS file CUSTOMER, which is a CREATK-defined file:

```
custread:
        proc options(main);

/* this program opens and reads from */
/* a previously-created MIDAS file CUSTOMER */
/* which has fixed-length records */

dcl customer file keyed sequential;
dcl pkey char(5);
dcl readvar char(70);
/* reads keys */
```

```
/* KVAR is used with KEYTO option and must be CHAR VARYING */

/* set up an on-unit to handle end of file  */

        on endfile(customer) begin;
            close file(customer);
            put skip list('End of file');
            stop;
            end;
        open file(customer) input;
        pkey = '2194G';
        read file(customer) into(readvar) key(pkey);
        put skip list(readvar);
        close file(customer);
        end;
```

When run, the following is printed at the terminal:

```
2194GSpectrographics          NWOR
```

This program could easily be modified to perform updates and/or additions to the file.

ERROR HANDLING

The most commonly encountered errors in the PL/I interface are KEY and RECORD errors. It is a good idea to include on-units for these conditions in your programs, as well as an ENDFILE on-unit for files opened for INPUT or UPDATE.

Key Errors

The KEY condition may be raised for several reasons, including:

- The record with the indicated KEY cannot be found during a READ or REWRITE.
- The program attempted to add a record with a KEY value that already exists in the file; duplicate keys are not allowed.
- A partial KEY value is used during a READ or REWRITE (the use of partial keys is not supported).
- The size used in declaring the KEY variable during a READ is smaller than the default size indicated for the key in the index subfile (default for PL/I-created files is 32 characters).
- There is no more room to add keys in the primary index subfile (very rare; see Section 15 if this happens).
- The key value is in the wrong format or was not specified

properly; for example, the key might have been written as a fixed decimal and you are trying to read it back as a character string. (The reverse of the situation is also true.)

As an aid in debugging, the ONKEY function (built-in) returns the value of the key that caused the KEY error. The following program contains KEY and ENDFILE on-units to trap errors that may occur while performing file reads:

```
readerr:
        proc options(main);

/* this program shows the use of on-units in trapping KEY errors
   during file reads */

dcl err_file file output;
/* err_file is for error messages */
dcl onkey builtin;
/* returns key value that caused error condition to be raised */
dcl badkey char(32) var;
dcl sample file keyed sequential;
dcl pkey char(32);
dcl readvar char(30) var;
dcl kvar char(32) var;

/* KVAR is used with KEYTO option and must be CHAR VARYING */

/* Set up on-unit to handle file read errors */

        on key(sample) begin;
/* KEY condition is a common error */
            badkey = onkey;
/* assign value of error-causing key to BADKEY and print it out */
            put skip file(err_file) list ('Bad key is:', badkey);
            goto pick_up;
            end;                                  /* end on-unit */

/* set up an on-unit to handle end of file  */

        on endfile(sample) begin;
            close file(sample);
            put skip list('End of file');
            stop;
            end;
        open file(sample) input;
        open file(err_file);

/* the first READ doesn't have to be a keyed one */

        read file(sample) into(readvar);
        put skip list(readvar);

/* now read with bad key */
```

```
        pkey = '0006';                          /* no such key */

/* this should trigger on-unit for KEY */

        read file(sample) into(readvar) key(pkey);
        put skip list(readvar);

/* control comes here in event of an error */

pick_up:
        read file(sample) into(readvar) key('0001');

/* read top of file */

        put skip list(readvar);
        close file(sample);
        close file(err_file);
        end;
end; •
/* run it */
OK, seg #readerr

first file record
first file record

OK,
```

Note

Both the KEY and the RECORD conditions are always enabled and cannot be disabled.

The RECORD Condition

The RECORD error condition is generally raised during a READ, WRITE or REWRITE operation when the size of the record in the data subfile does not match the size of the variable into which the record is being read or from which it is being written. The variable used may be too small or too large to accommodate the size of the data being read into it. PL/I demands that the record size and read-variable match exactly. In PL/I Subset G, the only way to avoid this error is to program carefully (and cross your fingers).

Other Possible Error Conditions

The UNDEFINEDFILE condition may be raised during an unsuccessful attempt to open a file. This can occur for a variety of reasons:

- The filename is misspelled when opening the file for INPUT or UPDATE. (Use of these access modes assumes the file exists.)
- An attempt is made to open a non-existent file for INPUT or UPDATE.
- Incomplete or conflicting attributes are specified during an OPEN or DECLARE FILE statement.
- Your version of PL/I has not been loaded with the proper MIDAS interface routines and/or proper libraries.
- Your compiler is malfunctioning.

In MIDAS files, if one of the segment subfiles is left open (due to program error or failure) an attempt to re-open the file will trigger this condition. Use CLOSE ALL (PRIMOS level command) to close the file before attempting to re-run the program.

Locked Records

If the program aborts and a record on the file remains locked, you must run MPACK to unlock the record as the record cannot unlock itself. Locking usually occurs when the user hits CTRL-P or BREAK immediately following a READ operation. The only way to tell if a record is locked is that when you attempt to read that record an ERROR condition is signalled and the program will fail. At this point, you should run the MPACK utility using the UNLOCK option. (See Section 12.)

SECTION 10

THE RPG INTERFACE

INTRODUCTION

RPG can be used to access MIDAS files just like any other type of file supported by Prime's RPG. Prime's RPG supports both keyed-index and direct access MIDAS files. In RPG, keyed-index MIDAS files are called Indexed files; they must be indicated with an "I" in column 32 of the File Description statement. (This section refers to the standard RPG coding "Specifications Sheet" as a "statement.") Direct access MIDAS files are specified as Direct files, with a "D" in column 32. COBOL calls these files INDEXED SEQUENTIAL and RELATIVE files respectively.

About This Section

This section assumes you know RPG and understand the terms and concepts related to RPG file-handling. The purpose of this section is not to teach you how to use RPG, but rather how to use RPG to access a MIDAS file. In this section, you'll find the basic information needed to correctly describe and operate on a MIDAS file with an RPG program.

LANGUAGE-DEPENDENT FEATURES

Secondary keys and secondary data are not supported by RPG, so a file written and updated by RPG programs should not have these features. However, RPG can access a MIDAS file that has secondary keys and secondary data, but only by primary key. Other RPG restrictions on MIDAS files are:

- The primary key in an Indexed file must be in the data record, but it does not have to be the first field in the record.
- The primary key in an Indexed file must be no more than 32 characters long.
- The primary key in a Direct file must be defined as single-precision floating-point, using the "S" data type option. This is done during template creation: see Section 2.
- The primary key in a Direct file is called the relative record number. It is not physically resident in the data record and is therefore not defined as part of the data record in an RPG program. The RPG program describes the record number as a standard numeric item. The only time you worry about defining the relative record number is during template creation when you define the primary key as a single- precision floating-point number (see above).

- Records cannot be deleted from a MIDAS file in RPG.

- Primary key values cannot be changed in MIDAS, so the only part of a MIDAS file that can be updated from an RPG program is the data record. If the data record contains a copy of all the key values for that record, be careful not to change the key values while updating, or you'll end up with severe discrepancies between the key values in the data file records and the key values in the secondary indexes. This applies only to MIDAS files that were built by an application program in another language or by KBUILD.

- MIDAS files can be accessed randomly by key, if they're Indexed, or by relative record number, if they are Direct files, but only if designated as Chained files.

Program Execution Requirements

To run an RPG program that accesses a MIDAS file, the following statements must be included in every LOAD sequence to properly load all the libraries:

```
LI RPGLIB  /* RPG library
LI KIDALB  /* MIDAS library
LI         /* FORTRAN library
 .
 .
 .
```

RPGLIB is the RPG library, KIDALB is the MIDAS library and the FORTRAN library is loaded with LI. For an example of a complete load sequence, see Compiling and Loading later in this section.

DESCRIBING A MIDAS FILE IN RPG

The first part of MIDAS file-handling in RPG is the definition process, which consists of correctly describing the MIDAS file to RPG via the File Description statement. Table 1Ø-1 shows how to define a keyed-index or a direct access MIDAS file. Where necessary, the individual attributes are described in further detail below.

Table 10-1. RPGII File Description Specifications For MIDAS Files

Attribute	Column(s)	What to Specify
• File Type	15	Input (I), Output (O) or Update (U) but not Display (D)
• File Designation	16	Primary (P), Secondary (S), Chained (C) or Demand (D): left blank if an Output file
• File Format	19	Fixed length (F)
• Record Length	24-27	The MIDAS data record length: includes primary key length
• Mode of Processing	28	Specify R for random, L for sequential within limits - for Indexed files only, or leave blank for sequential by key
• Key-Field Length	29-30	The length of the primary key in characters: for Indexed files only
• File Organization	32	Indexed or Direct (I or D)
• Key Col. Position	35-38	Column where the primary key starts in the data record - for Indexed files only
• Device	40-46	Must be disk (DISK)
• File Addition	66	To add records to a non-empty Indexed file, specify an A in column 66. The records need not be added in sequential order. To load records into an empty Indexed file, define the file as an Output file (with an "O" in column 15) and put a U or a blank in column 66. U implies an unordered load; where records may be input in random key order; a blank implies an ordered load.

Terms Used in File Description

Some of the attributes mentioned in Table 10-1 require a bit more explanation, as it may not be clear how they relate to MIDAS. This is just a brief explanation of how these terms apply to MIDAS file processing in RPG, so refer to a standard RPG text if you need further assistance.

File Type Specification: The following restrictions apply to MIDAS files as described in column 15 of the File Description statement:

The file type (column 15) can be one of these:

- I -- Input (for reading only) *
- O -- Output (for writing only)
- U -- Update (for reading and writing) *

* If an A occurs in column 66 of the File Description statement, new records may also be added to the file, if an ADD entry appears in column 16-18 of the Output statement.

For further information on what these file types mean, and the restrictions on them, see The RPG Programmer's Guide.

Notice that a MIDAS file cannot be described as a Display (D) file in RPG.

File Designation Restrictions: The legal file designations for a MIDAS file, as indicated in column 16 are:

- P -- Primary: main file from which records are read -- only one primary file should be specified per program. The Primary file can be opened for input or update.
- S -- Secondary: one or more files from which records are read after Primary file is processed, if "matching" is not specified in columns 61-62 of the Input statement. If matching is indicated, secondary files are processed by standard RPG matching algorithms. Secondary files can be Input or Update files, and they are processed in the order in which they appear in the File Description statements.

- C -- Chained: can be read randomly or loaded directly via the CHAIN operation code. Chained files can be opened for Input, Output or Update.

- D -- Demand: can be input or update files. Use the READ operation code in the Calculation statement to read from a Demand file. These files are processed sequentially by key.

File Addition Specifications: Records can be added either sequentially or randomly to an Indexed (MIDAS) file. Records cannot be added to a file opened for access under the sequential within limits processing mode, however. For an Indexed file opened for Input, Output, or Update, if an A is specified in column 66, new records can be added in any order. For an Indexed file opened for Output, a blank indicates a load to an initially empty file where the records must be supplied in key order; a U in column 66 indicates an unordered load to an initially empty file. It is more efficient to add records in sorted order (by primary key value), because then MIDAS doesn't have to sort them during the loading process.

Records can be initially loaded to Direct files by specifying the file as Output Chained: use an "O" in column 15, and a "C" in column 16. Records can be added to a Direct file that already contains entries by specifying the file as Update Chained: use a "U" in column 15 and a "C" in column 16. In either case column 66 of the File Description statement, as well as columns 16-18 of the Output statement, should be left blank when processing Direct files. Note that relative record number values cannot exceed the number of records pre-allocated during template creation.

Modes of Processing

The method in which a MIDAS file is to be processed is indicated in column 28 of the File Description Specifications statement. The three modes of processing allowed on MIDAS files are:

- L -- Sequential within limits (for Indexed files only)

- R -- (1) Random by relative record number (Direct files) or (2) random by key (Indexed files)

- blank -- Sequential by key

These modes of processing can be applied to MIDAS files depending on how these file are described in the File Designation column.

Primary, Secondary or Demand Files: can be accessed by non-random access methods only. The particular access method (mode of processing) depends on on the file's organization.

Organization	Mode of Processing
Direct	Sequential by relative record number
Indexed	Sequential by key, Sequential within limits

For Chained Files: MIDAS files declared as Chained files may be processed according to their organization. If the MIDAS file is Indexed, it can be accessed randomly by key. If the file is Direct, it can be accessed randomly by relative record number only. MIDAS files can only be accessed by key if they are defined as Chained files.

Demand Files: Demand files can be processed sequentially by key (applies to both Indexed and Direct files), or sequentially within limits (for Indexed files only). There are two methods of accomplishing limits processing: using the SETLL operation in a Calculation statement, or using a record address file (RAF). (Record address files are sequential files used to perform limits processing on Indexed files.) The first method specifies a lower limit for the primary key value with SETLL: the file can then be positioned to the proper record and can be processed from that point. The record address file method requires the creation of a separate "limits" file (sequential) which contains records that specify the "low" value from which to start processing, and the "high" value at which to stop processing. To use the record address file method, you must specify the name of this file in the Extensions statement (in columns 11-18), along with the name of the MIDAS file that is to be processed (columns 19-26). See The RPG Programmer's Guide for details.

FILE OPERATIONS

The operations that can be performed on a MIDAS file vary with the file description in the File Description statement. The standard file operations that can be performed on MIDAS files with an RPG program are:

- Reading records -- can be done sequentially or randomly by primary key or relative record number (Direct files). A read always implies a position operation, and implies a lock operation if the file is opened for Update. (See Positioning the File for further details.) Reads are done in the normal Input cycle, or by a READ or CHAIN operation in a Calculation statement.

- Adding records -- is the addition of new records to a non-empty file. For indexed files this is possible for Input, Output, or Update files where an A appears in column 66 of the File

Description statement, and when ADD appears in columns 16-18 of the Output statement. Refer to Updating Records for an explanation of adding new records to a Direct file.

- Loading records -- is the initial addition of records to an empty MIDAS file. This applies to both Indexed and Direct files.

- Updating records -- works only for files opened as Update files: an "update" is simply a read followed by a write. This applies to Indexed or Direct files. The only way to add entries to an existing Direct file is to use an update operation.

These operations are elaborated upon below.

Positioning the File

File position is thought of in terms of a "file pointer" which always points to some record in the file. The record to which the file pointer is positioned is called the "current record", or the current file position. Only a file designated as "Chained" (using a C in column 16 of the File Description), can be positioned to a specific record in the file. File position is established using a CHAIN operation in the Calculation Specifications statement. When a file is CHAINed, using a specific primary key value, the file pointer positions to the record with this key value. This record then becomes the current record and is read. If the file is opened for Update, the record will also be locked. However, if the file is a Direct file defined as Output Chained, the CHAIN operation causes a position only.

Demand files can also be positioned, but only indirectly. This can be done by SETLL or with a record address file. The file position is established in two steps: first by setting the lower limit for the primary key with the SETLL operation in a Calculation statement, or by RAF, and then performing a READ operation. The record positioned to will be the one whose primary key value matches the lower limit value, if one exists. If this primary key value does not exist, the record whose primary key value is greater than the lower limit specification becomes the current record.

Reading Records

MIDAS files can be read as part of the normal Input cycle, or as part of the Calculation cycle, depending on their designation. Files whose records are read as part of the Input cycle are declared as Primary or Secondary and are read sequentially. Files declared as Demand can also be read sequentially from beginning to end using a SETLL operation, or by using a record address file to set an initial file position.

The read occurs during the READ operation of the Calculation statement. Records read from files declared as Chained are read randomly by primary or relative record key value using the CHAIN operation of the Calculation statement. If the file is an Update file, the record will also be locked when positioned to and read.

Sequential Reads on Indexed Files: Indexed files read as a part of the normal Input cycle are read sequentially by key. Each record is read in primary key order, that is, in the order in which key values appear in the primary index subfile. After a record is read, RPG automatically advances the file pointer, making the next record (in primary key order) the current record. The next read operation then reads the current record, and again advances the file pointer to the next record, making it current. Files opened as primary or secondary files cannot be read randomly.

Demand files can be read sequentially within limits in the two ways described earlier: see Demand Files under Modes of Processing, above.

Random Reads: Random reads can be done only on files designated as Chained files. This applies both to Indexed and Direct files. Random reads are performed by using the CHAIN operation in the Calculation statement. This method can be used on both Indexed files and Direct files. The primary key or relative record number is supplied to MIDAS, and the next record whose primary key matches this value will be returned. The indicator for "no record found" should be set (in columns 54 and 55 of the Calculation statement), enabling the program to recover if there is no record in the file with that key value or a given record number.

Note

In Direct files, space is pre-allocated for every record upon template creation. If a legal record number is supplied in a CHAIN operation, but there is no corresponding record for that number, the record will be returned as blanks. Note that this is different from COBOL which returns a "record not found" error in this case. RPG does not treat a read of a non-existent record as an error, as long as the record number is within the pre-allocated limit on record numbers.

Adding Records

Records can be added to a MIDAS file through standard RPG output methods or by using the KBUILD utility. See Section 3 for details on KBUILD.

In Indexed files that contain entries, if column 66 of the File Description statement contains an A and columns 16-18 of the Output statement specify the ADD Operation, RPG can add records to the file. The file can be opened for Input, Output, or Update. This applies only to files that have been initially loaded, that is, files that already contain entries.

Direct access files do not support ADD because of their file structure: instead, addition of records is accomplished by an update. See Updating Records below.

Loading Records

To load an Indexed file, specify a U or a blank in column 66 of the File Description statement: U implies an unordered load whereas blank implies an ordered (sequential by key) load. Note that ADD is not entered in columns 16-18 of the Output statement during a load.

To load a Direct file, specify it as Output Chained and use the CHAIN operation in the Calculation statement to indicate the relative record number for each record to be loaded.

Updating Records

To update a MIDAS file record in RPG, the record must first be read and then updated in the Output cycle. Be careful not to change the primary key value when rewriting a record, because MIDAS does not allow the primary key value to be changed. This applies to both Indexed and Direct files.

The only way to add new records to a Direct file that contains entries is to perform an update. This is because RPG assumes that any record for which space has been pre-allocated (by CREATK), but which was not added during the initial load, exists as all blanks. Thus, any attempt to add a new entry to a direct access file is simply an update of the blanks that already appear in this "slot" in the file.

Deleting Records

There is no delete operation in RPG. In order to delete records, write a program to "flag" the records to be deleted, then copy the file to a new file, omitting the records which were flagged for deletion. For an idea of how this can be done, see the sample program called UPDATE under DIRECT FILE EXAMPLES.

Error Handling

RPG flags all MIDAS errors but can only recover from conditions of "record locked" and "record not on file". The messages are described below. The general format of the messages are shown here.

The general MIDAS error message is reported as:

MIDAS [MIDAS error message number] filename

(In some messages, the word MIDAS may be returned as "KIDA" instead, but the user should not be concerned, as the two terms mean the same thing.) When a message of this type is encountered during file processing, the user should look up the error message number in Appendix A of this book to determine the nature of the error. Note that MIDAS errors cannot be handled by typing an S followed by a carriage return. In general, such action should only be taken when the message returned by RPG explicitly permits this method of recovery.

Record Locked: The general "record locked" message is:

filename keyfield RECORD IN USE. TYPE S(CR) TO TRY AGAIN.

The record has been locked in anticipation of an update. Keyfield is the key that MIDAS was processing when the error occurred. If no other user is accessing the file, the file was not properly closed after previous usage. If another user is accessing the file, take the S(CR) option, otherwise close all files and perform the necessary steps, depending upon your application, to have a reliable file. If you type S followed by (CR), the operation is executed again.

CHAIN Errors: An unsuccessful CHAIN operation will invoke the message:

****UNSUCCESSFUL CHAIN TO ABOVE RECORD. TYPE S(CR) TO SKIP PAST OUTPUT.

This message is returned when a CHAIN operation has been attempted on the file and MIDAS was unable to retrieve the record. This is a serious message and indicates that the file is probably corrupted, or that the file is not a valid MIDAS file. Typing S(CR) restarts the RPG program cycle, skipping all the operations that would involve this record.

Read Errors: The message for a general read error occurring at PRIMOS level is returned as:

****UNSUCCESSFUL READ AT LINE nn.

This indicates an I/O error at the system level: users have no control over such errors.

MIDAS Concurrency Errors: The MIDAS concurrency error message which RPG users may encounter is returned through RPG as:

```
MIDAS CONCURRENCY ER 13, filename
key
```

key represents the key value which RPG was processing at the time the error occurred.

In a multi-user environment, it's possible for more than one user to access the same segment subfile (index) and get in each other's way. Usually, this message occurs when one user deletes a record that another user has locked for reading and/or update. Although this can't happen with two RPG users, it's conceivable that some FORTRAN or COBOL user may have the same file open for update while an RPG user is simply reading from it. The RPG user may get this mesage when attempting to update a now-deleted record.

INDEXED FILE EXAMPLES

To give you a general idea of how a MIDAS template can be created, populated and accessed through RPG, we've assembled the following examples:

- Creating the template for a keyed-index MIDAS file called MASTER
- Listing of a record-loading program that adds employee data records to the empty MASTER file
- Listing of a file-reading program which verifies that all desired records were loaded
- Loading the two programs
- Output from the read program -- a sample report

Creating the Template

The Indexed file MASTER is created using CREATK as shown in this sample session. The primary key is defined as an ASCII key of 10 characters in length. The data size is defined as 32 words (64 characters). User input is underlined to distinguish it from CREATK's prompts.

```
[CREATK rev 17.6]

MINIMUM OPTIONS? yes

FILE NAME? master
NEW FILE? yes
DIRECT ACCESS? no

DATA SUBFILE QUESTIONS

PRIMARY KEY TYPE: a
PRIMARY KEY SIZE = : b 1Ø
DATA SIZE = : 32

SECONDARY INDEX

INDEX NO.? (CR)

OK,
```

Because RPG does not support secondary keys, we didn't define any for this file template: instead, we hit a carriage return (CR).

Loading the File

The following program, LOAD, loads the file MASTER with employee records from a sequential disk file called SFILE. The primary key values are taken from the employee number values in the sequential data file records. Remember, only empty files can be loaded sequentially.

The "H"" that appears at the top of the program is a Header card, and is optional if no entries are to be included in it.

```
H
F*   LOAD
F*
F*       THIS PROGRAM LOADS THE INDEXED FILE MASTER WITH EMPLOYEE
F*       RECORDS FROM THE SEQUENTIAL DISK FILE SFILE.  THE EMPLOYEE
F*       NUMBER IS USED AS THE INDEX.
F*
F*
FSFILE   IPE F      64           DISK
FMASTER  O   F      64  5AI     2 DISK
ISFILE   NS  Ø1
I                                       2   6ØEMPNO
I                                       7  64 DATA
OMASTER  D       Ø1
O                                    1 'P'
O                        EMPNO       6
O                        DATA       64
```

The sequential file SFILE contains the following records:

```
04416124ADAMS  WJ01234567800MH  200 380000  32000
22236124BROWN  HY23456789000MH20000 800000  60000
25781125COOPER IG33344555502SH  450 750000  52000
39840124DAVIS  TV44455666601MH  250 400000  30000
47124123EVANS  AS34567890100MH350001400000 120000
66031123FOX    FL45678901200MH  350 660000  26000
73315125HOLMES EB56789012300MH10000 400000  30000
80081125JONES  CO00011222200SH  400 640000  64000
86789123KELLER ND99988777701MS  300 520000  38000
98570124LAKE   MP88877666604SS30000120000   80000
```

Reading an Indexed File

When the program above is run, the records in the sequential file are added to the MIDAS file MASTER . To read the records back from the file and print out a report showing what's in the file, we use the READ program, listed below. The program reads MASTER sequentially and prints out a report in the file PRINT.

```
H
F*  READ
F*
F*    THIS PROGRAM READS THE INDEXED MASTER FILE IN SEQUENTIAL ORDER
F*    AND PRODUCES AN EMPLOYEE LISTING REPORT FROM THE DATA.  TOTALS
F*    ARE CALCULATED FOR CERTAIN CATEGORIES AND ARE SHOWN IN THE
F*    REPORT.
F*
F*
FMASTER  IPEAF      64  5AI     2 DISK
FPRINT   O   F      96            PRINTER
IMASTER  NS  01   1 CP
I                                       2   60EMPNO
I                                       7    9 DEPT
I                                      10   16 NAME
I                                      17   18 INIT
I                                      19   27 SSN
I                                      28  290EXEM
I                                      30   30 MSTAT
I                                      31   31 PSTAT
I                                      32  362PRATE
I                                      37  432YTDG
I                                      44  502YTDT
I                                      62   62 DEL                 10
C   10      PSTAT     COMP 'H'                        20
C   10      YTDG      SUB  YTDT      NETPAY  72
C   10      TOTAL     ADD  NETPAY    TOTAL   82
C   10      GROSS     ADD  YTDG      GROSS   82
C   10      TAX       ADD  YTDT      TAX     82
C   10      PRATE     COMP 300.00               02
C   10 20   HOURLY    ADD  1         HOURLY  20
C   10N20   SALAR     ADD  1         SALAR   20
```

```
OPRINT   H    2   1P
O        OR       OF
O                          UDATE Y    8
O                                    47 'EMPLOYEE LISTING'
O                                    73 'PAGE'
O                          PAGE  Z   78
O         H              1P
O        OR             OF
O                                     8 'EMPLOYEE'
O                                    19 'EMPLOYEE'
O                                    32 'DEPARTMENT'
O                                    39 'RATE'
O                                    50 'Y-T-D'
O                                    62 'Y-T-D'
O                                    74 'Y-T-D'
O         H  2           1P
O        OR             OF
O                                     7 'NUMBER'
O                                    17 'NAME'
O                                    30 'NUMBER'
O                                    50 'GROSS'
O                                    61 'TAX'
O                                    72 'NET'
O         D              01 10
O                          EMPNO      7
O                          NAME      17
O                          INIT      20
O                          DEPT      29
O                          PRATE 1   40
O                          YTDG  1   52
O                          YTDT  1   64
O                          NETPAY1   76
O                     02                78 '*'
O         T 1            LR
O                                    14 'TOTAL SALARIED'
O                                    25 'EMPLOYEES:'
O                          SALAR 2   28
O                          GROSS 1   52
O                          TAX   1   64
O                          TOTAL 1   76
O         T              LR
O                                    12 'TOTAL HOURLY'
O                                    23 'EMPLOYEES:'
O                          HOURLY2   28
```

When compiled, loaded and executed, the program produces a report that looks like this:

9/23/80 EMPLOYEE LISTING PAGE 1

EMPLOYEE NUMBER	EMPLOYEE NAME		DEPARTMENT NUMBER	RATE	Y-T-D GROSS	Y-T-D TAX	Y-T-D NET	
04416	ADAMS	WJ	124	2.00	3,800.00	320.00	3,480.00	
22236	BROWN	HY	124	200.00	8,000.00	600.00	7,400.00	
25781	COOPER	IG	125	4.50	7,500.00	520.00	6,980.00	
39840	DAVIS	TV	124	2.50	4,000.00	300.00	3,700.00	
47124	EVANS	AS	123	350.00	14,000.00	1,200.00	12,800.00	*
66031	FOX	FL	123	3.50	6,600.00	260.00	6,340.00	
73315	HOLMES	EB	125	100.00	4,000.00	300.00	3,700.00	
80081	JONES	CO	125	4.00	6,400.00	640.00	5,760.00	
86789	KELLER	ND	123	3.00	5,200.00	380.00	4,820.00	
98570	LAKE	MP	124	300.00	12,000.00	800.00	11,200.00	
TOTAL SALARIED EMPLOYEES:2					71,500.00	5,320.00	66,180.00	
TOTAL HOURLY EMPLOYEES: 8								

DIRECT FILE EXAMPLES

Direct files require a slightly different type of template. The user must answer "yes" to the "DIRECT ACCESS?" question which CREATK always asks after requesting the file name and status. MIDAS then sets up the template for direct access, allowing each record to have, and be accessed by, a unique floating-point record number. The RPG user, when creating a template for a Direct file, must always define the PRIMARY KEY TYPE as "S", which is the symbol for single-precision floating-point keys in MIDAS in an RPG program. The primary key is never mentioned in the description of the data record. The example below shows the sample user-CREATK dialog needed to set up the Direct file MASTD used in the remaining examples in this section.

Creating a Direct File Template

The following CREATK session shows the input needed to define the Direct file MASTD, which is used in the remaining examples to demonstrate Direct file processing in RPG.

```
OK, creatk
[CREATK rev 17.6]

MINIMUM OPTIONS? yes

FILE NAME? mastd
NEW FILE? yes
DIRECT ACCESS? yes

DATA SUBFILE QUESTIONS

PRIMARY KEY TYPE: s
DATA SIZE = : 32
NUMBER OF ENTRIES TO ALLOCATE? 1ØØ

SECONDARY INDEX

INDEX NO.? (CR)

SETTING FILE LOCK TO N READERS AND N WRITERS

OK,
```

In this case, the file has fixed-length records of 32 words (64 characters). CREATK allocates 1ØØ records for this file when it sets up the template. Only record numbers between Ø and 99 are legal for this file.

Loading the File

The following program, DALOAD, loads an empty Direct file using a sequential disk file, FILEIN. The sequential file appears below the program listing.

```
H
F*  DALOAD
F*
F*    THIS PROGRAM LOADS THE DIRECT ACCESS FILE MASTD WITH EMPLOYEE
F*    RECORDS FROM THE SEQUENTIAL DISK FILE FILEIN.  THE RELATIVE
F*    RECORD NUMBERS OF THE FILE ARE SELECTED FROM THE FIRST TWO
F*    DIGITS OF THE EMPLOYEE NUMBER.
F*
F*
FFILEIN  IPE F      64            DISK
FMASTD   OC  F      64R   D       DISK
IFILEIN  NS  Ø1
I                                          2   3ØEMPNO
I                                          4   6ØEMP1
I                                          7  64 DATA
C   Ø1      EMPNO     CHAINMASTD                 99
OMASTD   D        Ø1
O                                      1 'P'
O                        EMPNO         3
O                        EMP1          6
O                        DATA         64
```

When compiled and loaded, the program is run to add records to the MASTD file.

Reading the File

The following program uses the just-loaded Direct file as a master file against which data in a transaction file can be checked. The transaction file is a sequential file containing employee record information. Records are read from the sequential file called TRANS and the MASTD file is then read to see if the information matches. Any data in the transaction file which does not match the data in the MASTD file is marked with an asterisk (*) in the edit listing produced by the program. Note that the relative record number does not appear as part of the record description.

```
H
F*  DAREAD
F*
F*  THIS PROGRAM CHECKS THE SEQUENTIAL TRANSACTION FILE FOR ERRORS
F*  AND PRODUCES A CURRENT EARNINGS EDIT LISTING.  TESTS ARE MADE
F*  FOR THE CORRECT TRANSACTION CODE, FOR THE CORRECT DEPARTMENT
F*  NUMBER, AND ALSO FOR A EMPLOYEE NUMBER MATCH WITH THE DIRECT
F*  ACCESS FILE MASTD.
F*
F*
FMASTD   IC  F      64R   D       DISK
FTRANS   IPE F      64            DISK
FLIST1   O   F     120            PRINTER
ITRANS   NS  01
I                                        1   1 CODE
I                                        8   90EMPNO
I                                       10  120EMPNO2
I                                       13  150DEPT
I                                       25  282RHOURS
I                                       29  322OHOURS
I                                       33  372GROSS
I                                       38  422TAX
IMASTD   NS  02   1 CP
I                                        2   30MEMPNO
I                                        4   60EMPNO2
C   01                  SETON                          30
C   01      DEPT        COMP 125                       20
C   01N20   DEPT        COMP 124                       20
C   01N20   DEPT        COMP 123                       20
C   01      EMPNO       CHAINMASTD                 21
C  N20 01   ERRDEP      ADD  1         ERRDEP  20
C   21 01   ERREMP      ADD  1         ERREMP  20
C   01      CODE        COMP 'C'                       22
C  N22 01   ERRRC       ADD  1         ERRRC   20
C   01      TOTRH       ADD  RHOURS    TOTRH   52
C   01      TOTOH       ADD  OHOURS    TOTOH   52
C   01      TOTG        ADD  GROSS     TOTG    62
C   01      TOTT        ADD  TAX       TOTT    62
C   01      TOTAL       ADD  1         TOTAL   20
OLIST1   H          1P
O       OR          OF
O                                      38 'CURRENT EARNIN'
O                                      53 'GS EDIT LISTING'
O        H 11       1P
O       OR          OF
O                                       4 'CODE'
O                                      12 'EMP NO'
O                                      21 'DEPT NO'
O                                      30 'REG HRS'
O                                      39 'OTM HRS'
O                                      48 'GROSS'
O                                      57 'TAX'
O        D 1        30
O                           CODE        3
```

```
O                          EMPNO      9
O                          EMPNO2    12
O                          DEPT      19
O                          RHOURS1   29
O                          OHOURS1   38
O                          GROSS 1   48
O                          TAX   1   58
O        D  1      30N22
O       OR         30 21
O       OR         30N20
O                 N22                 3 '*'
O                  21                10 '*'
O                 N20                18 '*'
O                                    76 'CHECK FOR ERRORS'
O        T 2       LR
O                                    21 'TOTALS'
O                          TOTRH 2   29
O                          TOTOH 2   38
O                          TOTG  2   48
O                          TOTT  2   58
O        T 1       LR
O                                    14 'ERROR RECORDS:'
O                                    27 'RECORD CODE-'
O                          ERRRC 2   29
O                                    47 'EMPLOYEE NUMBER-'
O                          ERREMP2   49
O                                    69 'DEPARTMENT NUMBER-'
O                          ERRDEP2   71
O        T 1       LR
O                                    15 'TOTAL RECORDS P'
O                                    24 'ROCESSED:'
O                          TOTAL 2   27
```

The sequential disk file TRANS, whose data records are checked for errors, contains the following information:

```
C       04416123          4000 500 9500  950
C       22236124          0000 00020000 1500
C       25781125          4000 00018000 1300
O       39840122          4000 00010000  750
C       47124123          0000 00035000 3000
C       66013123          4000 30015575  808
C       68726124          0000 00040000 4000
C       73315125          0000 00010000  750
D       74249123          0000 00025000 1750
C       80081152          3500 00014000 1400
C       86789123          4000 00012000  950
C       98570124          0000 00030000 2000
```

After the program has been compiled, loaded and run, the printer file, LIST1, looks like this:

CURRENT EARNINGS EDIT LISTING

CODE	EMP NO	DEPT NO	REG HRS	OTM HRS	GROSS	TAX	
C	04416	123	40.00	5.00	95.00	9.50	
C	22236	124	.00	.00	200.00	15.00	
C	25781	125	40.00	.00	180.00	13.00	
O *	39840	122 *	40.00	.00	100.00	7.50	CHECK FOR ERRORS
C	47124	123	.00	.00	350.00	30.00	
C	66031	123	40.00	3.00	155.75	8.08	
C	68726 *	124	.00	.00	400.00	40.00	CHECK FOR ERRORS
C	73315	125	.00	.00	100.00	7.50	
D *	74249 *	123	.00	.00	250.00	17.50	CHECK FOR ERRORS
C	80081	152 *	35.00	.00	140.00	14.00	CHECK FOR ERRORS
C	86789	123	40.00	.00	120.00	9.50	
C	98570	124	.00	.00	300.00	20.00	
	TOTALS		235.00	8.00	2,390.75	191.58	

ERROR RECORDS: RECORD CODE- 2 EMPLOYEE NUMBER- 2 DEPARTMENT NUMBER- 2

TOTAL RECORDS PROCESSED: 12

Updating Records

In Update mode, new records can be added to a file (if it already contains data), and existing data records can be rewritten. Updates to a Direct file can be done using a sequential file which tells the update program what to do with certain records in the Direct file. The sample program UPDATE shown below uses the file MAINT to indicate which records in the MASTD file will be updated, added or effectively "deleted". Records are not deleted physically, but appear to have been deleted when the report which UPDATE produces is printed out. The report is done to verify that all the proper alterations have been made to the file.

The file MAINT contains the following information:

```
MA23712123CARSON BG99977555500SH  300 470000  47000
MA70345125HARRIS CH44466888802MS32500115000O 105000
MD39840
MD80081
MD98570
MU22236124MORGAN  2MS20000
MU25781125COOPER 02SH  500
MU74249124IRVING 03MS25000
MU86789123KELLER  1MS15000
```

A listing of the UPDATE program begins on the next page.

```
H
F*  UPDATE
F*
F*  THIS PROGRAM UPDATES THE DIRECT ACCESS MASTD FILE VIA THE SEQUENTIAL
F*  MAINTENANCE FILE.  THIS RESULTS IN ADDITIONS, DELETIONS AND UPDATES.
F*  A REPORT IS GENERATED TO LIST THE CHANGES THAT WERE MADE TO THE
F*  MASTER FILE.
F*
F*
FMASTD   UC  F     64R   D      DISK
FMAINT   IPE F     64           DISK
FLIST2   O   F    120           PRINTER
IMASTD   NS       1 CP
I                                      2   30EMPNO
I                                      4   60EMPNO2
I                                      7   90DEPT
I                                     10  16 NAME
I                                     17  18 INIT
I                                     19  27 SSN
I                                     28  290EXEM
I                                     30  30 MSTAT
I                                     31  31 PSTAT
I                                     32  362PRATE
I                                     37  432YTDG
I                                     44  502YTDT
I                                     62  62 DEL
IMAINT   AA  01   1 CM   2 CA
I                                      3   40TEMPNO
I                                      5   70TEMP2
I                                      8  100TDEPT
I                                     11  17 TNAME
I                                     18  19 TINIT
I                                     20  28 TSSN
I                                     29  300TEXEM
I                                     31  31 TMSTAT
I                                     32  32 TPSTAT
I                                     33  372TPRATE
I                                     38  442TYTDG
I                                     45  512TYTDT
I        BB  02   1 CM   2 CD
I                                      3   40TEMPNO
I                                      5   70TEMP2
I        CC  03   1 CM   2 CU
I                                      3   40TEMPNO
I                                      5   70TEMP2
I                                      8  100TDEPT
I                                     11  17 TNAME
I                                     18  190TEXEM
I                                     20  20 TMSTAT
I                                     21  21 TPSTAT
I                                     22  262TPRATE
C  N01      TEMPNO    CHAINMASTD              H1
C   01      TEMPNO    CHAINMASTD              40
C   03      DEPT      COMP TDEPT                  11
```

```
C   Ø3      NAME      COMP TNAME                    12
C   Ø3      EXEM      COMP TEXEM                    13
C   Ø3      MSTAT     COMP TMSTAT                   14
C   Ø3      PSTAT     COMP TPSTAT                   15
C   Ø3      PRATE     COMP TPRATE                   16
C   Ø1      ADDS      ADD  1         ADDS    2Ø
C   Ø2      DELS      ADD  1         DELS    2Ø
C   Ø3      CHNGES    ADD  1         CHNGES  2Ø
OLIST2   H 1      1P
O       OR        OF
O                                    43 'EMPLOYEE MASTER'
O                                    6Ø 'FILE MAINTENANCE'
O        H 1      1P
O       OR        OF
O                                     1 'C'
O                                     6 'EMP'
O                                    12 'DEPT'
O                                    24 'FM'
O                                    39 'NO'
O                                    42 'M'
O                                    45 'S'
O                                    77 'DE'
O        H  2     1P
O       OR        OF
O                                     1 'D'
O                                     5 'NO'
O                                    11 'NO'
O                                    18 'NAME'
O                                    24 'II'
O                                    35 'SOC SEC NO'
O                                    39 'EX'
O                                    42 'S'
O                                    45 'H'
O                                    53 'RATE'
O                                    64 'YTD GROSS'
O                                    73 'YTD TAX'
O                                    77 'CD'
O        D  1     Ø1
O                          TEMPNO     4
O                          TEMP2      7
O                          TDEPT     11
O                          TNAME     2Ø
O                          TINIT     24
O                          TSSN      34
O                          TEXEM 1   39
O                          TMSTAT    42
O                          TPSTAT    45
O                          TPRATE1   53
O                          TYTDG 1   64
O                          TYTDT 1   74
O                                    88 'NEW RECORD'
O        D  1     Ø2
O                                     1 'P'
O                          EMPNO      4
```

```
O                       EMPNO2     7
O                       DEPT      11
O                       NAME      20
O                       INIT      24
O                       SSN       34
O                       EXEM  1   39
O                       MSTAT     42
O                       PSTAT     45
O                       PRATE 1   53
O                       YTDG  1   64
O                       YTDT  1   74
O                                 77 'D'
O                                 84 'DELETE'
O        D  1     03
O                                  1 'P'
O                       EMPNO      4
O                       EMPNO2     7
O                       TDEPT     11
O                N11               12 '*'
O                       TNAME     20
O                N12               21 '*'
O                       INIT      24
O                       SSN       34
O                       TEXEM 1   39
O                N13               40 '*'
O                       TMSTAT    42
O                N14               43 '*'
O                       TPSTAT    45
O                N15               46 '*'
O                       TPRATE1   53
O                N16               54 '*'
O                       YTDG  1   64
O                       YTDT  1   73
O                                 84 'CHANGE'
O        T 1      LR
O                                 14 'RECORDS ADDED-'
O                       ADDS  2   16
O                                 40 'RECORDS INACTIVATED-'
O                       DELS  2   42
O                                 62 'RECORDS CHANGED-'
O                       CHNGES2   64
OMASTD   D        01 40
O                                  1 'P'
O                       TEMPNO     6
O                       TDEPT      9
O                       TNAME     16
O                       TINIT     18
O                       TSSN      27
O                       TEXEM     29
O                       TMSTAT    30
O                       TPSTAT    31
O                       TPRATE    36
O                       TYTDG     43
O                       TYTDT     50
```

```
O                                        62 ' '
O       D        Ø2
O                                        62 'D'
O       D        Ø3
O                N11     TDEPT           9
O                N12     TNAME          16
O                N13     TEXEM          29
O                N14     TMSTAT         3Ø
O                N15     TPSTAT         31
O                N16     TPRATE         36
```

When run, the UPDATE program produces this report:

EMPLOYEE MASTER FILE MAINTENANCE

C D	EMP NO	DEPT NO	NAME	FM II	SOC SEC NO	NO EX	M S	S H	RATE	YTD GROSS	YTD TAX	DE CD	
	23712	123	CARSON	BG	999775555	ØØ	S	H	3.ØØ	4,7ØØ.ØØ	47Ø.ØØ		NEW RECORD
	7Ø345	125	HARRIS	CH	444668888	Ø2	M	S	325.ØØ	11,5ØØ.ØØ	1,Ø5Ø.ØØ		NEW RECORD
P	3984Ø	124	DAVIS	TV	444556666	Ø1	M	H	2.5Ø	4,ØØØ.ØØ	3ØØ.ØØ	D	DELETE
P	8ØØ81	125	JONES	CO	ØØØ112222	ØØ	S	H	4.ØØ	6,4ØØ.ØØ	64Ø.ØØ	D	DELETE
P	9857Ø	124	LAKE	MP	888776666	Ø4	S	S	3ØØ.ØØ	12,ØØØ.ØØ	8ØØ.ØØ	D	DELETE
P	22236	124	MORGAN *	HY	234567890	Ø2*	M	S*	2ØØ.ØØ	8,ØØØ.ØØ	6ØØ.ØØ		CHANGE
P	25781	125	COOPER	IG	333445555	Ø2	S	H	5.ØØ*	7,5ØØ.ØØ	52Ø.ØØ		CHANGE
P	86789	123	KELLER	ND	999887777	Ø1	M	S	15Ø.ØØ*	5,2ØØ.ØØ	38Ø.ØØ		CHANGE

RECORDS ADDED- 2 RECORDS INACTIVATED- 3 RECORDS CHANGED- 3

SECTION 11

DIRECT ACCESS

INTRODUCTION

Direct access MIDAS files are supported by the FORTRAN, COBOL and RPGII interfaces. However, such files are particularly well-suited for use with COBOL, which treats them as standard RELATIVE files. This section explains the structure of a direct access file and how to create a direct access file template. Since each of the three language interfaces to MIDAS that support direct access files do things a little differently, we make no attempt to lump them all together. Instead, only the COBOL direct access interface is discussed in this section. For information on using direct access in FORTRAN and in RPG, see sections 6 and 10 of this book.

What Is Direct Access?

Direct access in MIDAS is based on record numbers; each record in the file has a unique floating-point record number (single-precision) that identifies that record absolutely. To get a particular record, one simply provides MIDAS with the right record number and the record is found and returned. There's no need to search through many index blocks in search of a value (which may have mistakenly gotten inserted in the wrong spot), because direct access involves calculating the exact physical location of the record in the file using an algorithm that takes into account the record size, segment size, and so forth. The user doesn't need to worry about this except to know that MIDAS takes care of it all internally. The only drawback to direct access is that the user must keep track of the correlation between record numbers and record values in order to locate individual records.

Direct Access File Structure

A direct access file template is created with CREATK in much the way as are keyed-index MIDAS files. In fact, the dialogs used are virtually identical, except for a few prompts, which are shown in the sample dialog a bit later in this section. The basic differences between keyed-index and direct access MIDAS files are:

- A direct access (RELATIVE) file must have fixed-length records; the user must supply the record length (data size) in words.
- Each record in a direct access file must have a unique record number. The record number may or may not have to be the primary key: this restriction depends on the language interface being used to access the file.

- Storage space must be pre-allocated for a direct access file, which means that the user must estimate the maximum number of entries which will eventually reside in the data subfile. CREATK then allocates the proper amount of space required to accommodate a file with the number of records the user indicated.

- Direct access MIDAS files take up a bit more disk space than keyed-index files because a two-word record number is stored with each primary index entry. (This is true even when the primary index is defined as the record number.)

As mentioned in section 1, direct access files are accessible by the MIDAS direct access method, but they can also be accessed by the keyed-index method if keys are included in the file template. This applies to FORTRAN only.

DIRECT ACCESS IN EACH LANGUAGE

The three language interfaces that support the direct access feature of MIDAS each implement it a bit differently. The following paragraphs summarize the treatment of direct access in COBOL, FORTRAN and RPG.

Direct Access in COBOL

COBOL treats direct access MIDAS files as RELATIVE files, using the standard RELATIVE file I/O statements, with a few minor enhancements, as COBOL's interface to direct access MIDAS files. The RELATIVE KEY in a RELATIVE file is the primary key you defined for that MIDAS file during template creation. It is a COBOL requirement that the RELATIVE KEY be the relative record number field in each record: therefore, the primary key for a direct access MIDAS file to be accessed as a COBOL RELATIVE file must be defined in a special way.

Defining the Primary Key (in CREATK): From the COBOL programmer's point of view, the primary key would be thought of as a character string from one to six characters (bytes) in length. However, you must specify the primary key in bit string form when creating a direct access template with CREATK. This means that the primary key is always defined as being from 8 bits to 48 bits in length: this corresponds to a character string with a minimum size of one character and a maximum size of six characters. This allows for a maximum of 999,999 entries in the file, because 999999 is the largest relative record number possible in a RELATIVE file under Prime's COBOL.

Declaring the RELATIVE KEY in the Program: A corresponding PICTURE clause must be used to define the RELATIVE KEY in any program that accesses a RELATIVE file: for example, a 48-bit string would have a PICTURE clause of 9(6).

Observe that reducing the RELATIVE KEY size from 48 bits decreases the maximum number of entries that the file can accommodate. For example, if the KEY is defined as a 32-bit string, the file can have a maximum of 9999 entries, as opposed to 999,999 entries. The PICTURE clause that would describe this particular key in a program is PIC 9(4). In a COBOL program that accesses a RELATIVE file, the RELATIVE KEY cannot be declared as part of the data record; instead, it is declared in the WORKING STORAGE section.

Secondary keys are not supported in RELATIVE files because COBOL provides no mechanism for adding entries to secondary index subfiles. These points are explained further under RELATIVE FILE ACCESS, below.

Direct Access in FORTRAN

Direct access files in FORTRAN do not require that the record number be defined as a primary or secondary key. However, the user may define the record number as the primary or a secondary key if desired, keeping in mind that the record number is stored by MIDAS as a single-precision floating-point number. If the user does not want the record number to be a key field, the primary key should be defined as some other unique field in the record. In addition, up to 17 secondary indexes may be defined during template creation.

If you choose not to make the record number a key field, you don't have to worry about it at all during template definition. The only time you need to be concerned about the record number is when adding entries to the file. Then, a unique record number must be supplied for each record to be added to the data subfile. MIDAS takes care of storing the record numbers in the proper place.

Direct access files can be built (populated) by KBUILD, as described in Section 3; record numbers must be supplied by the user and must appear in the same word position in each record. To access direct access files by record number (instead of by key), the same basic subroutine calls are used, but the communications array format is slightly different. Details on access methods are covered in Section 6.

Direct Access in RPG

RPGII supports direct access MIDAS files as standard RPG direct files. When a direct access MIDAS file template is created specifically for use with RPG, the primary key must be defined as a single-precision floating-point number. Specify "S" in response to the "PRIMARY KEY TYPE:" prompt of the CREATK dialog. In RPG programs that process direct access files, the file organization must be specified as Direct (D) and the file can only be opened as a Chained file. Records are read randomly by record number. See Section 10 for specifics.

CREATING A DIRECT ACCESS FILE

The CREATK dialog for setting up a direct access template is very much the same as the dialog used to set up a keyed-index file template. To invoke the dialog, simply type CREATK, and answer "YES" to the "DIRECT ACCESS?" prompt. The important parts of the dialog are discussed below, and several examples are provided to illustrate how it is used. In creating a direct access file, you may choose either the minimum options path by answering "YES" to the MINIMUM OPTIONS?" prompt, or the extended options path, which is described in Section 15.

CREATK Dialog for Direct Access

The dialog used to set up a direct access template under minimum options is explained below. Most of the prompts are identical to those already shown in Section 2.

Prompt	Response
MINIMUM OPTIONS?	YES.
FILENAME?	Enter pathname of file to be created.
DIRECT ACCESS?	Enter YES: signifies that file is to be set up for direct access and that it can be accessed by record number: records will be stored in the data subfile in sequential order by record number.
DATA SUBFILE QUESTIONS	
PRIMARY KEY TYPE:	For files to be used with COBOL, enter "b" for bit string; for files used with RPG, specify "s"; for FORTRAN, any of the key types supported by CREATK are okay. (See Table 2-2 in Section 2.)
PRIMARY KEY SIZE=:	For COBOL, specifiy a bit string from 8 to 48 bits; for FORTRAN, replies should be made in accordance with the data type specified above. (See Section 2.)
DATA SIZE=:	Supply a value other than zero: direct access files must have fixed-length records. Do not simply hit (CR) in response to this prompt. Record size should be supplied in words.
NUMBER OF ENTRIES TO ALLOCATE?	Enter maximum number of entries (records) for which to reserve room in the data subfile. CREATK must pre-allocate space for a direct access file.
SECONDARY INDEX INDEX NO?	Enter an index number from 1-17 if secondary keys are desired. This feature applies to FORTRAN only; COBOL or RPG users should simply hit carriage return (CR), as a response to this prompt.

The remaining prompts in the dialog are the same as those of the keyed-index dialog discussed in section 2.

Sample CREATK Session

The following is taken from a terminal session in which a direct access template was created for a MIDAS file to be used with COBOL.

```
OK, creatk
[CREATK rev 17.6]

MINIMUM OPTIONS? yes

FILE NAME? dacust
NEW FILE? yes
DIRECT ACCESS? yes

DATA SUBFILE QUESTIONS

PRIMARY KEY TYPE: b
PRIMARY KEY SIZE = : b 48
DATA SIZE = : 35
NUMBER OF ENTRIES TO ALLOCATE? 15

SECONDARY INDEX

INDEX NO.? (CR)
SETTING FILE LOCK TO N READERS AND N WRITERS
OK,
```

Because we want to use the file in a COBOL application, no secondary keys were defined for this file template.

RELATIVE FILE ACCESS

As stated earlier, COBOL's method of accessing direct access files is the standard COBOL RELATIVE file interface. It consists of the READ, WRITE, REWRITE, DELETE and START statements. The standard OPEN and CLOSE statements are used to open and close the file from a COBOL program as shown in Section 7.

The remainder of this section deals with COBOL's RELATIVE file I/O statements and how they are used in reading, writing, rewriting and deleting records from a direct access MIDAS file. Because they are so similar in syntax and usage to the INDEXED SEQUENTIAL statements, the reader is often referred to Section 7 for explanations to avoid duplication.

Defining the File

The SELECT statement, discussed in Section 7, defines the file's logical name and organization. The differences between the SELECT statement for an INDEXED SEQUENTIAL file and a SELECT for a RELATIVE file are the ORGANIZATION clause specification and the the terms used to describe the primary key. The format indicates these differences:

```
SELECT filename

   ASSIGN TO PFMS

     ORGANIZATION IS RELATIVE

     [ACCESS MODE IS { SEQUENTIAL }
                     { RANDOM     } ]
                     { DYNAMIC    }

     [RELATIVE KEY IS key-name-1]

   [FILE STATUS IS status-code].
```

RELATIVE KEY is the primary key and represents the record number in a direct access file. It should be defined as a bit string during template creation. However, when accessing the file through a program, the RELATIVE KEY is treated as a character string with a minimum size of one character and a maximum size of six characters. This RELATIVE KEY should not be included in the DATA RECORD description in the File Definition section of the program. Instead, it should be defined only in the WORKING STORAGE section. The key need not be specified if the file is opened for SEQUENTIAL access. (See Access Modes below.)

Basically, all the rules that govern the RECORD KEY definition in INDEXED SEQUENTIAL files apply to RELATIVE files, with the following additions:

- The RELATIVE KEY is a character string defined as a bit string during template definition and must represent the record number in each file record.
- The RELATIVE KEY cannot have a value larger than 999,999 or a PICTURE clause larger than 9(6).
- The RELATIVE KEY named by key-name-1 must be defined in the Working Storage section, and not in the Record Description for that file.

Access Modes: The access modes are the same as for INDEXED SEQUENTIAL files; see Section 7 for details. The correspondence between access modes and "open" modes and what can be done in each situation is shown in Table 11-1. If the file is opened for SEQUENTIAL OUTPUT, it is not necessary to define the RELATIVE KEY in the program because it is supplied by the COBOL run-time library. If you do not define a key in the SELECT statement, it defaults to a 6-byte character string, which is the maximum size allowed.

The status codes returned in status-code are listed in Table 11-2.

Condition Handling

The INVALID KEY and AT END clauses may be included in RELATIVE file I/O statements as indicated in the formats given here. These statements are identical in format to those used in INDEXED file handling, explained in Section 7. The status codes returned to the program during RELATIVE file processing are listed in Table 11-2. Be sure to handle all common file processing conditions like "end-of-file" and "record not found" and so forth.

Here is a summary of the condition handlers that can be used to trap run-time errors and conditions during RELATIVE file processing:

- The AT END clause, which defines one or more statements which are executed when an "end-of-file" is detected. This statements or statements may or may not direct control to some "end-of-file-handling" statements or procedure at another location in the program.

- The INVALID KEY clause, which is executed when an error occurs; it specifies a statement or series of statements which perform some useful action or series of actions in the event of a KEY error. This statements or series of statements may or may not direct control to another location in the program where the error can be handled appropriately. (Can be used in START, READ, WRITE, REWRITE and DELETE statements.)

- The USE AFTER ERROR statement which indicates the name of a procedure in the program which will be executed in the event of an I-O error, in addition to the System's standard I-O procedures.

Table 11-1. Statements Permitted in Each Access Mode

RELATIVE I/O

File Access Mode	Statement	Open Mode		
		Input	Output	Input-Output
Sequential	READ	●		●
	WRITE		●	
	REWRITE			●
	START	●		●
	DELETE			●
Random	READ	●		●
	WRITE		●	●
	REWRITE			●
	START			
	DELETE			●
Dynamic	READ	●		●
	WRITE		●	●
	REWRITE			●
	START	●		●
	DELETE			●

Table 11-2. RELATIVE File Status Codes

Status Code	Interpretation
ØØ	Successful completion of operation.
1Ø	End of file encountered during a READ.
21	User has attempted to write beyond predefined boundaries of the file.
22	Record already exists in data subfile; user attempted to add a record with a non-unique record number.
23	Record not found; no record found with the specifed key value.
24	Boundary violation: user has attempted to read or write beyond pre-allocated file boundaries. (Boundaries are allocated by CREATK during template creation.)
3Ø	Permanent I/O error; could be a parity error, data check or transmission error.
(Status Codes beginning with 9 are Prime-defined condition codes.)	
9Ø	Locked record; attempt to access a record already locked by another user or process.
91	Unlocked record; REWRITE attempted without first locking the record with a READ.
94	MIDAS concurrency error: another user has deleted the record you were trying to access.
95	User has supplied a record length for a RELATIVE file that does not match the record size assigned to the file during template creation.
96	Relative record number error; user supplied a record number larger than the number pre-allocated by CREATK.

98 Attempt to do an indexed add to a direct access file; can't add entries to a RELATIVE file even if it's opened for INDEXED access.

99 System error; possibly serious. Before panicking, verify that error is not due to a START that encountered a locked record.

Opening and Closing the File

A direct access file is opened using the OPEN statement:

```
OPEN { INPUT  }
     { OUTPUT } filename
     { I-O    }
```

Direct access files may be opened for INPUT, OUTPUT or I-O:

- INPUT means READ only.
- OUTPUT means WRITE only.
- I-O means all operations are legal including READ, UPDATE, DELETE and WRITE.

In SEQUENTIAL access mode, a RELATIVE file cannot be opened for I-O if a WRITE statement is included for this file; instead, it must be opened for OUTPUT only. Only empty files can be opened for SEQUENTIAL OUTPUT. OUTPUT mode prevents the accidental movement of the record pointer while COBOL is trying to add records in sequential record order, because the only operation legal in OUTPUT mode is a WRITE. COBOL provides its own record numbers in SEQUENTIAL WRITEs, ignoring any values the user might have supplied for the relative record number field.

Note

When a RELATIVE file is opened for writing in SEQUENTIAL access mode, it must be OPENed for OUTPUT: it cannot be opened for I-O. The file is assumed to be empty (that is, it contains no entries).

Closing the File: The file is closed using the CLOSE statement:

```
CLOSE filename-1 [,filename-2...]
```

Put the CLOSE statement at the logical end of the program or in an error-handling routine. It's possible to close more than one file with a single CLOSE statement.

ADDING RECORDS TO A RELATIVE FILE

Because RELATIVE files in COBOL do not support the use of secondary keys, adding a record to a direct access file consists of simply supplying the record number and the data to be added to the data subfile. It is not possible to add secondary index entries to a RELATIVE file.

Two Ways to Add Records

Because of their structure, RELATIVE files must be handled a little differently than INDEXED files when populating them. If a file is empty (contains no record or index subfile entries), you can add records to that file in any of the three access modes. However, once a file contains entries, you can no longer add records to it in SEQUENTIAL access mode. This is because SEQUENTIAL access mode is specifically intended for intial loading of records, that is, for populating empty files. RANDOM and DYNAMIC access modes are most useful for inserting records anywhere in a file that already has entries. Both RANDOM and DYNAMIC access modes allow users to supply their own record numbers for each record to be added to the file.

The WRITE Statement

The WRITE statement is used, in all three access modes, to add records to a RELATIVE file. There are two methods of record addition possible: the first, available in SEQUENTIAL mode, lets COBOL supply the record numbers for each record -- the user just supplies the data record information. This is known as "loading" or "initially loading" a file. The file must be empty in order to be loaded. The "random" method, available in the other two access modes, requires that the user supply a record number for each record added. In this type of record addition, there are "slots" pre-allocated for them in the data subfile. In other words, each record will get put in the proper slot according to its assigned record number.

Sequential Record Addition

In SEQUENTIAL access mode, the file must be opened for OUTPUT, not I-O, as mentioned previously in the note following Opening and Closing the File. The other restriction on sequential record addition is that once the file contains entries, you can't add records to it in SEQUENTIAL mode: you must use RANDOM or DYNAMIC access mode. In SEQUENTIAL access mode, the user does not supply record numbers for each entry added, but instead lets COBOL take care of providing a unique number for each record. COBOL always starts numbering with the lowest possible number in the sequence which happens to be 000001, if the RELATIVE KEY is declared as 6 characters (48 bits), and is incremented by 1 each time a new record is added. Although it's not illegal for the user to provide record numbers during sequential record addition they are ignored by COBOL anyway. In fact, if the RELATIVE KEY field is defined by the user in the program (this is not required, as mentioned above), COBOL will return, in the RELATIVE KEY field, the record number of each record you add after each WRITE operation is complete.

Note

Follow this rule when adding records to a RELATIVE file: once a file contains entries, whether they were added sequentially or randomly, the file should not be opened for SEQUENTIAL OUTPUT. New records can only be added to the file if it is opened for DYNAMIC or RANDOM access.

The format of the WRITE statement is:

```
WRITE record-name [FROM new-record]
[INVALID KEY imperative-statement].
```

record-name is the name of a record description associated with the file in the DATA DIVISION of the program. The "FROM new-record" clause is optional; without it, the user must explicitly MOVE the new record information to record-name so it can be written to the file.

For Example: The example below shows how records can be added interactively to a RELATIVE file. (User input is underlined to distinguish it from program output.)

```
       ID DIVISION.
       PROGRAM-ID. ADD-PROG.
       AUTHOR. LJD.
       INSTALLATION. TPUBS.
       DATE-WRITTEN. 09/02/80.
       DATE-COMPILED. 09/02/80.
       SECURITY. NONE.
       REMARKS. PROGRAM TO TEST DACUST FILE ADDS.
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       SOURCE COMPUTER. PRIME.
       OBJECT COMPUTER. PRIME.
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT DACUST ASSIGN TO PFMS
               ORGANIZATION IS RELATIVE
               ACCESS MODE IS SEQUENTIAL
               RELATIVE KEY IS RECORD-NO
               FILE STATUS IS STATUS-CODE.
       DATA DIVISION.
       FILE SECTION.
       FD  DACUST
           LABEL RECORDS ARE STANDARD
           VALUE OF FILE-ID IS "DACUST"
           DATA RECORD IS CUST-FILE-RECORD.
       01  CUST-FILE-RECORD.
           02  CUST-ID PIC X(5).
           02  CUST-NAME  PIC X(25).
           02  LOCATION-CODE  PIC X(4).
           02  FILLER PIC X(35).
```

```
WORKING-STORAGE SECTION.
Ø1  STATUS-CODE PIC 99 VALUE ZERO.
Ø1  READ-REC PIC X(35) VALUE SPACES.
Ø1  RECORD-NO PIC 9(6) VALUE ZEROES.
PROCEDURE DIVISION.
FIRST-PROC.
    OPEN OUTPUT DACUST.
GET-DATA.
    DISPLAY 'ENTER DACUST ID - - PIC X(5)  OR ENTER XX TO QUIT'.
    ACCEPT CUST-ID.
    IF CUST-ID = 'XX' GO TO FINIS.
GET-REST.
     DISPLAY 'ENTER CUST-NAME -- 25 CHARS'.
     ACCEPT CUST-NAME.
     DISPLAY 'ENTER LOCATION CODE -- 4 CHARS (REGION-STATE)'.
     ACCEPT LOCATION-CODE.
     WRITE CUST-FILE-RECORD  INVALID KEY GO TO CHECK.
     GO TO GET-DATA.
CHECK.
     IF STATUS-CODE NOT = ØØ DISPLAY 'RECORD NOT ADDED'
      ELSE GO TO GET-REST.
     GO TO GET-DATA.
FINIS.
     IF STATUS-CODE = ØØ DISPLAY 'ALL DONE'
      ELSE
     DISPLAY 'STATUS CODE  IS:'  STATUS-CODE.
     CLOSE DACUST.
     STOP RUN.
```

```
ok, seg #add.da
ENTER FILE ASSIGNMENTS:
> /
FILE ASSIGNMENTS COMPLETE.
ENTER DACUST ID - - PIC X(5)  OR ENTER XX TO QUIT
4456P
ENTER CUST-NAME -- 25 CHARS
LESLEY'S PASTE-UP SHOP
ENTER LOCATION CODE -- 4 CHARS (REGION-STATE)
NEMA
ENTER DACUST ID - - PIC X(5)  OR ENTER XX TO QUIT
3378T
ENTER CUST-NAME -- 25 CHARS
AUTOMAT TYPESETTERS
ENTER LOCATION CODE -- 4 CHARS (REGION-STATE)
MWID
ENTER DACUST ID - - PIC X(5)  OR ENTER XX TO QUIT
XX
ALL DONE
```

When read back sequentially, the records will be returned in the order added. See Reading Sequentially, below.

Note

START operations are not legal in RELATIVE files opened for SEQUENTIAL access in OUTPUT mode -- so there's no way to start adding records at any point in the file other than the beginning of the file.

Adding Records Randomly

In random writes, the user must supply a record number for each record added. The format is the same as that shown for sequential reads, above. The file can be opened for OUTPUT or I-O.

Example: This program opens a file for I-O in DYNAMIC access mode and does random, interactive adds. Naturally, records can be added by reading in the data from another disk file as well, but a record number must be supplied with each record added. The important thing to note about adding records randomly is that record numbers must be supplied in the proper form -- don't forget those leading zeroes!

```
ID DIVISION.
PROGRAM-ID. ADD-PROG.
AUTHOR. LJD.
INSTALLATION. TPUBS.
DATE-WRITTEN. 09/02/80.
DATE-COMPILED. 09/02/80.
SECURITY. NONE.
REMARKS. PROGRAM TO TEST DACUST FILE WRITES.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE COMPUTER. PRIME.
OBJECT COMPUTER. PRIME.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT DACUST ASSIGN TO PFMS
        ORGANIZATION IS RELATIVE
        ACCESS MODE IS DYNAMIC
        RELATIVE KEY IS RECORD-NO
        FILE STATUS IS STATUS-CODE.
DATA DIVISION.
FILE SECTION.
FD  DACUST
    LABEL RECORDS ARE STANDARD
    VALUE OF FILE-ID IS "DACUST"
    DATA RECORD IS CUST-FILE-RECORD.
01  CUST-FILE-RECORD.
    02  CUST-ID PIC X(5).
    02  CUST-NAME  PIC X(25).
    02  LOCATION-CODE  PIC X(4).
    02  FILLER PIC X(35).
WORKING-STORAGE SECTION.
01  STATUS-CODE PIC 99 VALUE ZERO.
01  READ-REC PIC X(35) VALUE SPACES.
```

```
       Ø1  RECORD-NO PIC 9(6) VALUE ZEROES.
       PROCEDURE DIVISION.
       FIRST-PROC.
           OPEN OUTPUT DACUST.
       GET-DATA.
            DISPLAY 'ENTER RECORD NUMBER'.
            ACCEPT RECORD-NO.
           DISPLAY 'ENTER DACUST ID - - PIC X(5) OR ENTER XX TO QUIT'.
           ACCEPT CUST-ID.
           IF CUST-ID = 'XX' GO TO FINIS.
       GET-REST.
            DISPLAY 'ENTER CUST-NAME -- 25 CHARS'.
            ACCEPT CUST-NAME.
            DISPLAY 'ENTER LOCATION CODE -- 4 CHARS (REGION-STATE)'.
            ACCEPT LOCATION-CODE.
            WRITE CUST-FILE-RECORD  INVALID KEY GO TO FINIS.
            GO TO GET-DATA.
       FINIS.
            IF STATUS-CODE = ØØ DISPLAY 'ALL DONE'
             ELSE
            DISPLAY 'STATUS CODE  IS:'  STATUS-CODE.
            CLOSE DACUST.
            STOP RUN.
```

OK,

Some sample input to the program might be (user input is underlined):

```
ENTER RECORD NUMBER
ØØØØØ5
ENTER DACUST ID - - PIC X(5) OR ENTER XX TO QUIT
5556X
ENTER CUST-NAME -- 25 CHARS
MISCELLANEOUS SUPPLIERS
ENTER LOCATION CODE -- 4 CHARS (REGION-STATE)
SEGA
ENTER RECORD NUMBER
ØØØØØ6
ENTER DACUST ID - - PIC X(5) OR ENTER XX TO QUIT
5556A
ENTER CUST-NAME -- 25 CHARS
ROCKY POINT ART SHOP
ENTER LOCATION CODE -- 4 CHARS (REGION-STATE)
NENY

etc.
```

You can add entries in any order. Don't try to write a record that already exists and don't try to write beyond the pre-allocated file boundaries. If you only allocated 15 records, then don't try to add a record with number ØØØØ16 or above. Remember that CREATK asks for a number of records for which to allocate space for in an index subfile.

READING A RELATIVE FILE

There are two types of READ formats for RELATIVE files: the sequential read format and the random read format. The formats are similar to those used in reading INDEXED SEQUENTIAL files.

Sequential Reads

In a direct access file opened for SEQUENTIAL access, the READ operation retrieves records in order by record number. The initial record number value with which to begin is established by a MOVE and START, by setting an initial value in WORKING STORAGE, or by default, in which the initial position is set to the first record in the file. The first file record by definition has the lowest record number in the entire file. Sequential READs are generally associated with SEQUENTIAL access mode, although they are possible in DYNAMIC mode also.

The READ statement format used for sequential retrieval is:

```
READ filename [NEXT RECORD] [INTO read-var]
[AT END imperative-statement].
```

filename is the name of the RELATIVE file. The "INTO read-var" clause causes the record read to be moved from the record buffer associated with the file into the read-var variable. The NEXT clause is used only in DYNAMIC access mode; it is not necessary if the file is opened for SEQUENTIAL access mode because a READ operation automatically causes the file pointer to move to the next record in the file. The "AT END" clause is required unless there is a USE AFTER procedure under the DECLARATIVES for handling errors that occur while processing the file.

Example: The following program reads values from a RELATIVE file sequentially by record number. Note that the file is opened for INPUT, which means that the file is opened for reading only, as opposed to writing and/or update.

```
       ID DIVISION.
       PROGRAM-ID. DACUST-TST.
       AUTHOR. LJD.
       INSTALLATION. TPUBS.
       DATE-WRITTEN. 09/02/80.
       DATE-COMPILED. 09/02/80.
       SECURITY. NONE.
       REMARKS. PROGRAM TO TEST DACUST FILE READS.
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       SOURCE COMPUTER. PRIME.
       OBJECT COMPUTER. PRIME.
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT DACUST ASSIGN TO PFMS
               ORGANIZATION IS RELATIVE
               ACCESS MODE IS SEQUENTIAL
```

```
            RELATIVE KEY IS RECORD-NO
            FILE STATUS IS STATUS-CODE.
        DATA DIVISION.
        FILE SECTION.
        FD  DACUST
            LABEL RECORDS ARE STANDARD
            VALUE OF FILE-ID IS "DACUST"
            DATA RECORD IS CUST-FILE-RECORD.
        Ø1  CUST-FILE-RECORD.
            Ø2  CUST-ID PIC X(5).
            Ø2  CUST-NAME  PIC X(25).
            Ø2  LOCATION-CODE.
                Ø5  REGION      PIC XX.
                Ø5  STATE       PIC XX.
        WORKING-STORAGE SECTION.
        Ø1  STATUS-CODE PIC 99 VALUE ZERO.
        Ø1  RECORD-NO PIC 9(6) VALUE ZEROES.
        Ø1  READ-REC PIC X(35) VALUE SPACES.
        PROCEDURE DIVISION.
        FIRST-PROC.
            OPEN INPUT DACUST.
            MOVE LOW-VALUES TO RECORD-NO.
            START DACUST KEY IS NOT LESS THAN RECORD-NO INVALID KEY
             GO TO END-FILE.
        READ-LOOP.
            READ DACUST  INTO READ-REC AT END GO TO END-FILE.
            DISPLAY 'RECORD NO. IS:' RECORD-NO.
            DISPLAY READ-REC.
            GO TO READ-LOOP.
        END-FILE.
             CLOSE DACUST.
             IF STATUS-CODE =ØØ DISPLAY 'SUCCESSFUL COMPLETION.'
             ELSE
             DISPLAY 'STATUS CODE  IS:'  STATUS-CODE.
             STOP RUN.
```

When run, the program output is:

```
RECORD NO. IS:ØØØØØ1
C4456PLESLEY'S PASTE-UP SHOP   NEMA
RECORD NO. IS:ØØØØØ2
3378TAUTOMAT TYPESETTERS      MWID
SUCCESSFUL COMPLETION.
OK,
```

Reading the Current Record: In DYNAMIC access mode, READ without the KEY IS or NEXT RECORD clauses returns the current record. In RANDOM access mode, READ without the KEY IS clause also returns the current record. (Remember that sequential reads are not possible in RANDOM mode.) current record is the one just read or positioned to by a START operation (START is not legal in RANDOM access). In SEQUENTIAL access mode, the file pointer is advanced automatically to the next record in the file each time a READ statement is encountered.

Keyed Reads

Keyed reads (random reads) are permitted in DYNAMIC and RANDOM access modes, using the RELATIVE KEY. To do a keyed read, simply MOVE the appropriate record number value to the RELATIVE KEY field, then use this form of the READ statement:

```
READ filename [INTO read-var]
[INVALID KEY imperative-statement].
```

The "INTO read-var" option is used to move the data record from the buffer into which it is read to some program-specified temporary storage area (named by read-var) so the user can do something with it like DISPLAY, for instance. The INVALID KEY clause is required unless a USE AFTER procedure under the DECLARATIVES specifies what to do when errors occur during processing of that file.

This format requires that a new value be moved to the RELATIVE KEY field before each READ, or the same record will be returned over and over again. Remember, the only way to do a keyed READ on a RELATIVE file is to supply a record number value for the RELATIVE KEY field.

Keyed READ Example: The following example shows an excerpt from a program that does random READs on the DACUST file.

```
PROCEDURE DIVISION.
FIRST-PROC.
    OPEN INPUT DACUST.
    MOVE 000002 TO RECORD-NO.
READ-RANDOM.
    READ DACUST INTO READ-REC
    INVALID KEY GO TO KEY-ERR.
    DISPLAY 'RECORD NUMBER IS:' RECORD-NO.
    DISPLAY READ-REC.
READ-CUR.
    READ DACUST INTO READ-REC
     INVALID KEY GO TO KEY-ERR.
    DISPLAY  'KEY FIELD NOT UPDATED:'.
    DISPLAY 'READ RETURNS CURRECT RECORD.'.
    DISPLAY READ-REC.
    GO TO FINIS.
KEY-ERR.
     DISPLAY 'STATUS CODE  IS:'  STATUS-CODE.
FINIS.
     IF STATUS-CODE = 00 DISPLAY 'ALL DONE'.
     CLOSE DACUST.
     STOP RUN.
```

When run, the program prints the following:

```
RECORD NUMBER IS:000002
3378TAUTOMAT TYPESETTERS      MWID
KEY FIELD NOT UPDATED:
READ RETURNS CURRENT RECORD.
3378TAUTOMAT TYPESETTERS      MWID
ALL DONE
OK,
```

Reading the Current Record: In DYNAMIC and RANDOM access modes, if the RELATIVE KEY field is not updated between the last READ or START and the current READ, a READ without the NEXT RECORD clause will return the current record; that is, the record just read or positioned to by a START operation. See the above example.

UPDATING RECORDS

The REWRITE statement simply replaces the current record with a new text string, completely destroying the original. It's essential to remember that REWRITE does not establish or alter file position: therefore, you must do a READ before a REWRITE in order to tell MIDAS which record is to be updated and to lock the record. This is true in all access modes. Remember also that the file must be opened for I-O in order to update it.

The REWRITE format is:

```
REWRITE record-name [FROM new-record]
[INVALID KEY imperative-statement].
```

record-name is the name of a record associated with a file described in the File Description under the File Section of the program. The "FROM new-record" clause is optional because you can always move the new record value to record-name before you do the REWRITE. The INVALID KEY clause is mandatory in RANDOM and DYNAMIC modes, unless the DECLARATIVES includes a USE AFTER procedure for dealing with errors that occur while processing this file. The INVALID KEY clause should not be included in REWRITE statements when the file is opened for SEQUENTIAL access.

Example: This program opens the DACUST file for DYNAMIC access and updates an existing record:

```
ID DIVISION.
PROGRAM-ID. UPDATES.
AUTHOR. LJD.
INSTALLATION. TPUBS.
DATE-WRITTEN. 09/02/80.
DATE-COMPILED. 09/02/80.
SECURITY. NONE.
REMARKS. PROGRAM TO TEST DACUST FILE UPDATES.
```

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE COMPUTER. PRIME.
OBJECT COMPUTER. PRIME.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT DACUST ASSIGN TO PFMS
        ORGANIZATION IS RELATIVE
        ACCESS MODE IS DYNAMIC
        RELATIVE KEY IS RECORD-NO
        FILE STATUS IS STATUS-CODE.
DATA DIVISION.
FILE SECTION.
FD  DACUST
    LABEL RECORDS ARE STANDARD
    VALUE OF FILE-ID IS "DACUST"
    DATA RECORD IS CUST-FILE-RECORD.
01  CUST-FILE-RECORD.
    02  CUST-ID PIC X(5).
    02  CUST-NAME  PIC X(25).
    02  LOCATION-CODE.
        05  REGION      PIC XX.
        05  STATE       PIC XX.
WORKING-STORAGE SECTION.
 01  STATUS-CODE PIC 99 VALUE ZERO.
 01  READ-REC PIC X(35) VALUE SPACES.
 01  RECORD-NO PIC 9(6) VALUE ZEROES.
 01  NEW-RECORD  PIC X(35) VALUE SPACES.
PROCEDURE DIVISION.
FIRST-PROC.
    OPEN I-O DACUST.
    MOVE 000001 TO RECORD-NO.
READ-THIS.
    READ DACUST INTO READ-REC
      INVALID KEY GO TO KEY-ERR.
    DISPLAY 'CURRENT RECORD IS:' READ-REC.
 CHANGE-VAL.
    MOVE '2334PSEACOAST FINISHERS       WRCA' TO NEW-RECORD.
 UPDATE.
      REWRITE CUST-FILE-RECORD FROM NEW-RECORD
      INVALID KEY GO TO KEY-ERR.
      READ DACUST INTO READ-REC INVALID KEY
         GO TO KEY-ERR.
      DISPLAY 'UPDATED RECORD IS:' READ-REC.
      GO TO FINIS.
KEY-ERR.
     DISPLAY 'STATUS CODE  IS:'  STATUS-CODE.
FINIS.
     IF STATUS-CODE = 00 DISPLAY 'ALL DONE'
     ELSE DISPLAY 'STATUS CODE IS:' STATUS-CODE.
     CLOSE DACUST.
     STOP RUN.
```

```
OK, seg #up.da
ENTER FILE ASSIGNMENTS:
> /
FILE ASSIGNMENTS COMPLETE.
CURRENT RECORD IS:4456PLESLEY'S PASTE-UP SHOP    NEMA
UPDATED RECORD IS:2334PSEACOAST FINISHERS        WRCA
ALL DONE
OK,
```

DELETING RECORDS

The DELETE statement removes an indicated record from a direct access file. Deletes can be done in any access mode: RANDOM, DYNAMIC or SEQUENTIAL, but the file must be opened for I-O in all cases. The DELETE format is:

DELETE filename RECORD [INVALID KEY imperative-statement].

In RANDOM and DYNAMIC modes the INVALID KEY clause must be used unless a USE AFTER procedure for trapping errors is included under the DECLARATIVES. The INVALID KEY clause is not legal in SEQUENTIAL access mode because the record to be deleted is established by a READ and not by the DELETE operation itself.

Deletes in SEQUENTIAL Mode

In SEQUENTIAL access mode the record to be deleted must be READ before a DELETE can be executed. This is because the DELETE statement in SEQUENTIAL acess mode cannot establish a current record position and must instead depend on a READ to do so. The INVALID KEY clause should not be included in the DELETE statement when the file is opened for SEQUENTIAL access.

Deletes in DYNAMIC and RANDOM Modes

In DYNAMIC and RANDOM access modes, it is not necessary to do a READ prior to a DELETE because a DELETE establishes the current record position on its own. The user must supply a value for the RELATIVE KEY before a DELETE which is then used to position to that record before deleting it.

DELETE Examples

The first program excerpt shown below is from a program in which a RELATIVE file has been opened for I-O in SEQUENTIAL access mode. In order to delete a record from this file, the record number must be MOVEd to the RELATIVE KEY field, then read, and then deleted:

```
PROCEDURE DIVISION.
FIRST-PROC.
    OPEN I-O DACUST.
    MOVE 000001 TO RECORD-NO.
    START DACUST KEY IS NOT LESS THAN RECORD-NO INVALID KEY
     GO TO KEY-ERR.
    READ DACUST INTO READ-REC AT END GO TO FINIS.
      DISPLAY READ-REC.
 DEL-REC.
      DELETE DACUST.
```

The program prints out the record to be deleted just before the DELETE statement is actually executed. If you want to keep track of the records being deleted from a file, opening the file in SEQUENTIAL access mode forces you to read each record first, which makes it easy to keep an "audit file" of the records deleted.

When a RELATIVE file is opened for DYNAMIC access, the record to be deleted does not have to be read first and no START statement is required either, as this program excerpt shows:

```
PROCEDURE DIVISION.
FIRST-PROC.
    OPEN I-O DACUST.
    MOVE 000002 TO RECORD-NO.
 DEL-REC.
      DELETE DACUST INVALID KEY GO TO KEY-ERR.
```

Part IV
Maintenance and Administration

SECTION 12

MAINTAINING A MIDAS FILE

INTRODUCTION

MIDAS file maintenance generally involves monitoring MIDAS file index subfile and data subfile usage, and making minor adjustments to increase file efficiency. Maintenance also involves the periodic restructuring of the index subfiles and the data subfile with the MPACK utility to reclaim space "wasted" by data entries and index subfile entries which have been marked for deletion. Occasionally, records in the data subfile may become locked during use due to program abort or error, and the MPACK utility must be run to unlock these records.

This section discusses the CREATK options which can be used to monitor MIDAS file use and size, and the MPACK utility which can help you maintain an efficiently organized MIDAS file.

EXAMINING THE TEMPLATE

CREATK has other uses besides template creation. This section discusses the PRINT, USAGE, SIZE and VERSION options of CREATK. CREATK calls them "functions" -- we use functions and options interchangeably in referring to them. These functions provide details on file size, template structure, index usage and can estimate the number of segements needed, given a hypothetical number of entries for a data subfile. The CREATK options which can be used to modify a file are dealt with in Section 14.

Option Summary

To examine an existing file template, enter CREATK, provide the file's name or pathname when requested, and enter "no" to the "NEW FILE?" prompt as indicated in the example below. CREATK then prompts you to enter a function in response to the "FUNCTION?" prompt. To obtain a complete list of options, type "HELP". A list of functions and a brief description of each one is then displayed. Minimum abbreviations for each function are indicated with brackets; the portion enclosed in brackets is optional.

```
OK, creatk
[CREATK rev 17.6]

MINIMUM OPTIONS? yes

FILE NAME? customer
NEW FILE? no

FUNCTION? help
  A[DD]       =  ADD AN INDEX
  D[ATA]      =  CHANGE DATA RECORD SIZE
  E[XTEND]    =  CHANGE SEGMENT & SEGMENT DIRECTORY LENGTH
  F[ILE]      =  OPEN A NEW FILE
  H[ELP]      =  PRINT THIS SUMMARY
  M[ODIFY]    =  MODIFY AN EXISTING SUBFILE
  P[RINT]     =  PRINT DESCRIPTOR INFORMATION
  Q[UIT]      =  EXIT CREATK
  (C/R)       =  IMPLIED QUIT
  S[IZE]      =  DETERMINE THE SIZE OF A FILE
  U[SAGE]     =  DISPLAY CURRENT INDEX USAGE
  V[ERSION]   =  MIDAS DEFAULTS FOR THIS FILE
```

The options available for examining the template are:

PRINT — Prints description of various index structures.

USAGE — Displays number of entries in each index and the version of MIDAS last used to modify the file.

SIZE — Estimates number of segments and disk records needed to accommodate a given number of data file entries.

VERSION — Displays version of MIDAS used to create file.

The minimum abbreviation for each option is underlined. The function of each option is expanded upon in the following paragraphs.

The PRINT Option

PRINT displays index and data subfile information. It automatically prompts for an index subfile number with this prompt:

```
INDEX NO?
```

Enter a number from Ø to 17 if you want to see information on a particular index subfile. Enter the word "DATA" to examine data subfile information.

If an index subfile is being examined, CREATK displays:

- Number of segments allocated
- Index capacity (approximate number of entries that can be accommodated)
- Key type and size
- Number of index levels as of last MPACK.

For each index level in a particular index subfile, the PRINT option of CREATK displays this information:

- Block size (number of words per block)
- Key length
- Number of control words
- Maximum number of entries per block
- Length of an index entry
- Number of blocks in that level

For Example: The next example shows how PRINT (abbreviated "p") can be used to obtain information about all of the indexes in a MIDAS file. In this case, the statistics shown pertain to the CUSTOMER file. Note that the size for character type keys is displayed in both words and bytes. Since there is only one level of indexing for this file, (it contains only 5 entries) statistics are displayed for the "last level" of indexing only. In order to obtain the latest information on a file's index levels and number of blocks at each level, the file must have been MPACKed after the latest round of changes made to the file. See Section 15, Index Levels, for details on levels of indexing in the subfiles.

```
FUNCTION? print

INDEX NO.? Ø
1Ø SEGMENTS ALLOCATED WHICH CAN HOLD ABOUT  774144. ENTRIES
KEY TYPE:  CHARACTER
KEY SIZE     5 BYTES    (     3 WORDS)
# LEVELS:     1     SYNONYM ENTRIES NOT SUPPORTED

LAST LEVEL
    ENTRY SIZE:    6 WORDS    BLOCK SIZE: 1Ø24 WORDS    # CONTROL WORDS: 1Ø
    MAX ENTRIES/BLOCK:  169    # BLOCKS THIS LEVEL:   1.


FUNCTION? p

INDEX NO.? 1
1Ø SEGMENTS ALLOCATED WHICH CAN HOLD ABOUT  285696. ENTRIES
KEY TYPE:  CHARACTER
KEY SIZE    25 BYTES    (    13 WORDS)
# LEVELS:     1     SYNONYM ENTRIES SUPPORTED

LAST LEVEL
    ENTRY SIZE:   16 WORDS    BLOCK SIZE: 1Ø24 WORDS    # CONTROL WORDS: 1Ø
    MAX ENTRIES/BLOCK:   63    # BLOCKS THIS LEVEL:   1.


FUNCTION? p

INDEX NO.? 2
1Ø SEGMENTS ALLOCATED WHICH CAN HOLD ABOUT  926231. ENTRIES
KEY TYPE:  CHARACTER
KEY SIZE     4 BYTES    (     2 WORDS)
# LEVELS:     1     SYNONYM ENTRIES SUPPORTED

LAST LEVEL
    ENTRY SIZE:    5 WORDS    BLOCK SIZE: 1Ø24 WORDS    # CONTROL WORDS: 1Ø
    MAX ENTRIES/BLOCK:  2Ø2    # BLOCKS THIS LEVEL:   1.
```

If the "data" option is specified, CREATK displays:

- File type (keyed-index or direct-access)
- Number of index subfiles defined
- Number of entries currently indexed as of last MPACK
- Entry size (record size)
- Primary key size

```
FUNCTION? p

INDEX NO.? d  ("d" for data)
DATA SUBFILE:
   FILE TYPE: KI    # INDEXES:  3   # ENTRIES:      Ø.
   ENTRY SIZE    35 WORDS
   PRIMARY KEY SIZE:     5 BYTES (   3 WORDS)
```

Note that the DATA option returns the file configuration (FILE TYPE:) as either KI (keyed-index) or DA (direct access).

The SIZE Option

The SIZE option determines the number of segments and disk records required for an index subfile, a data subfile, or an entire file, given a projected number of records for a MIDAS file. The user supplies the number of records in response to this prompt:

```
NUMBER OF ENTRIES:
```

CREATK then asks for an index number:

```
INDEX NO:
```

to which the user should respond with one of the following:

- A number from Ø-17 -- obtain estimate for an individual index subfile
- DATA -- obtain an estimate for a data subfile
- TOTAL -- obtain an estimate for the entire file, including all index subfiles and data subfiles
- (CR) -- terminates the SIZE option dialog and returns you to the "FUNCTION?" prompt.

If the user specifies an index number or the DATA option, CREATK returns the following:

- Number of disk records needed for the index or data subfile
- Number of segments required to contain these index blocks
- Number of segments currently allocated for the index blocks already in the index or data subfile

```
FUNCTION? size
NUMBER OF ENTRIES: 400      (try 400 records )

INDEX NO.? 0
INDEX  0:        8 440 WD. RECS,        5 1024 WD. RECS
    2 SEGMENTS REQUIRED, 10 SEGMENTS ALLOCATED

INDEX NO.? 1
INDEX  1:       18 440 WD. RECS,        9 1024 WD. RECS
    2 SEGMENTS REQUIRED, 10 SEGMENTS ALLOCATED

INDEX NO.? 2
INDEX  2:        7 440 WD. RECS,        4 1024 WD. RECS
    2 SEGMENTS REQUIRED, 10 SEGMENTS ALLOCATED

INDEX NO.? data
DATA    :       37 440 WD. RECS,       16 1024 WD. RECS
     1 SEGMENTS REQUIRED, 327 SEGMENTS ALLOCATED

INDEX NO.? (CR)
```

Should the MIDAS file have variable length records, you'll get this message when you use the "data" option:

```
VARIABLE LENGTH DATA, NO COMPUTATION
```

If the TOTAL option is specified in response to the INDEX NO.? prompt, CREATK prints all the above information for each index subfile and the data file plus the number of disk blocks needed to accommodate all index subfiles and the data subfile. For example:

```
FUNCTION? size
NUMBER OF ENTRIES: 2000
INDEX NO.? total
INDEX  0:       30 440 WD. RECS,       14 1024 WD. RECS
    2 SEGMENTS REQUIRED, 10 SEGMENTS ALLOCATED
INDEX  1:       78 440 WD. RECS,       35 1024 WD. RECS
    2 SEGMENTS REQUIRED, 10 SEGMENTS ALLOCATED
INDEX  2:       26 440 WD. RECS,       12 1024 WD. RECS
    2 SEGMENTS REQUIRED, 10 SEGMENTS ALLOCATED
DATA    :      182 440 WD. RECS,       79 1024 WD. RECS
     1 SEGMENTS REQUIRED, 327 SEGMENTS ALLOCATED

TOTAL DISK RECORDS:      316 440 WD. RECS,      140 1024 WD. RECS
```

The USAGE Option

The USAGE option helps determine how many entries you've got in each index subfile. It displays the following information:

- Data records indexed as of last MPACK
- Data records added since last MPACK
- Data records deleted since last MPACK
- Total number of data records (entries) in the file
- The version of MIDAS which last modified the file

This example shows how many entries are currently in the primary and secondary index subfiles of a certain MIDAS file:

```
FUNCTION? usage

INDEX? Ø

ENTRIES INDEXED:       7
ENTRIES INSERTED:      Ø
ENTRIES DELETED:       2
TOTAL ENTRIES IN FILE:      5

LAST MODIFIED BY MIDAS REV. 17.6

FUNCTION? u

INDEX? 1

ENTRIES INDEXED:       7
ENTRIES INSERTED:      Ø
ENTRIES DELETED:       Ø
TOTAL ENTRIES IN FILE:      7

LAST MODIFIED BY MIDAS REV. 17.6

FUNCTION? u

INDEX? 2

ENTRIES INDEXED:       7
ENTRIES INSERTED:      Ø
ENTRIES DELETED:       Ø
TOTAL ENTRIES IN FILE:      7

LAST MODIFIED BY MIDAS REV. 17.6
```

The VERSION Option

The VERSION option displays the following information:

- The version of KIDALB (the MIDAS library) which was used in building the template
- The default DAM file length (524288 words)
- The default segment directory length (512 segments)
- Maximum segments allocated per index (10)
- The maximum number of indexes (including the primary) which can be defined for the file (18)

```
FUNCTION? v
[CREATK rev 17.6]

FILE CREATED BY KIDALB REV. 17.6

DEFAULT PARAMETERS FOR FILE
DAM FILE LENGTH       524288 WORDS
BASIC SEGMENT DIRECTORY LENGTH  512 SEGMENTS
MAXIMUM SEGMENTS PER INDEX  10
MAXIMUM NUMBER OF INDEXES  18
```

When debugging MIDAS applications, remember to check the version of MIDAS you're using -- it may help you to pinpoint the problem.

THE MPACK UTILITY

When you delete an index subfile entry, MIDAS automatically recovers the space formerly occupied by that entry. However, this is not the case for data subfile records: they are marked for deletion but are not physically removed until the MPACK utility is run. MPACK's chief function is to recover the space occupied by data records marked for deletion, freeing that space for additional records and increasing file efficiency -- but it does a lot of other things as well, like unlocking locked records and restructuring index subfiles.

Note

When making changes to the template structure with the ADD, DATA, EXTEND and MODIFY options of CREATK (covered in Section 14), you must MPACK the file after you've specified the changes. The desired modifications do not take effect until MPACK is executed on the file.

MPACK's Functions and Options

MPACK's specific functions are summarized below:

- Reclamation of space occupied by "deleted" data records
- Packing up of indexes to minimize disk space used
- Unlocking of all data records left locked after program abort or failure
- Reordering the data subfile to match the order of primary index subfile entries (complete file restructure)
- Logging errors and milestones to keep tabs on errors and to monitor the ongoing operation

Milestone/Error Logging: MPACK allows you to open an error/log file to keep track of any errors that occur during unlock or MPACK; in addition, an optional milestone status report can be recorded for a given number of records processed by MPACK. The milestone count is a user-specified number. The milestone statistics include date, time (in 24-hr. format), CPU and disk time used and the time expired between the current milestone and the previous one. For an example of a milestone event, see the sample output at the end of this section.

THE MPACK DIALOG

The MPACK dialog offers you two basic choices:

- You can restructure the file

  or

- You can simply unlock the locked file entries

To restructure index subfiles or the entire file, use the MPACK option, optionally abbreviated to M. The MPACK option then asks what part of the file is to be restructured. These options are described below. To perform the unlock operation, type UNLOCK or U.

The UNLOCK Option

The UNLOCK option simply goes through the data subfile entries looking for locked records. All locked records are unlocked and MPACK prints out a total count of unlocked records. The index subfiles are not touched during the "UNLOCK" option path.

The Restructure Option (MPACK)

The "MPACK" option path lets you restructure one or more index subfiles, all of the index subfiles, or the entire file. During an index restructure operation, MPACK searches the index subfile entries for entries that have been marked for deletion or for entries that are out of order. If any keys are out of sequence, MPACK reports them to the user, but does not reorder them. Secondary index entries which point to data subfile entries which have been marked for deletion are deleted so their space can be re-used.

In a data subfile restructure, the entries are reordered to correspond to the primary index order, and space wasted by "deleted" records is reclaimed for use. Space in the data subfile is reclaimed by copying the old data subfiles to new ones. At the end of the MPACK run, the old subfiles are deleted and the new ones replace them. Because of this copying procedure, the user must make sure that MPACK has enough disk space to make a copy of the original file being MPACKed.

Quickly summarized, the restructure options are:

- The index-number option: you simply specify which individual index subfiles are to be restructured.
- The DATA option: restructures the data subfile and all indexes.
- The ALL option: reclaims wasted space from all the index subfiles.

The DATA Sub-option: To restructure the entire file, use the DATA option. The chief benefit of the DATA option is that it not only checks and reorders the index subfile entries, but it also reorders the data subfile entries to correspond to the order of key entries in the primary index subfile and recovers data subfile spaces. In other words, it sorts the data subfile by primary key, using the key order in the primary index subfile. This makes sequential file processing a lot faster and makes key searches more efficient.

If the DATA option is specified, MPACK then asks if you want to overwrite the existing file or make a copy and work on that.

It is highly recommended that you always say NO to the "OK TO OVERWRITE?" query. This ensures that you will always have a copy of the original file in case you need it. In this case, MPACK asks you to specify the name of a file to which you want the restructured file to be written. The original file will be left intact and all the changes will be made to a copy of the file. If you accidentally specify the name of an existing file, (MIDAS or non-MIDAS), MPACK will display a message like:

```
FILENAME ALREADY EXISTS - TRY AGAIN.
```

This should prevent accidental overwrite of existing files and possible retribution by their owners.

The ALL Sub-option: The ALL sub-option of the MPACK option simply packs all the existing index subfiles, but does not touch the data subfile.

What If MPACK Aborts

If the MPACK process aborts for any reason, the original file is left unchanged, regardless of whether you are working on the old file or a "new" copy of it. If you're not overwriting the old file, but restructuring a copy of it, the new file will not be in very good shape, so you probably want to delete it. If overwriting an old file when an error or abort occurs, use KIDDEL and the "JUNK" option to delete all the partially used "scratch" space which MPACK uses during restructure.

The MPACK Dialog

The MPACK dialog is shown below with annotations. Step numbers have been included for clarity.

MPACK Dialog Prompts	User Responses
1. ENTER MIDAS FILENAME:	Enter pathname of existing MIDAS file.
2. MPACK OR UNLOCK?	Enter M(PACK) or U(NLOCK). If MPACK, dialog skips to step 3. If UNLOCK, dialog skip to step 6.
(MPACK Option)	Enter one of the following:

Note

In order to do an MPACK on a file, you need to have enough space for two copies of the file. MPACK always makes a copy of the file to work on so that in case of error, the original stands a lesser chance of being damaged.

IMPORTANT

3. ENTER LIST OF INDEXES, ALL OR DATA:	index number(s): to be MPACKed, separated by commas or spaces; dialog continues at prompt 6. A(LL): restructures all indexes in file and unlocks all data records; dialog resumes at step 6. D(ATA): restructures data

	file and indexes; dialog continues with prompt 4.
(DATA Option)	
4. OK TO OVERWRITE FILE?	Answer YES or NO. If YES, file is merely restructured and replaced (the original vanishes). If NO, the next prompt appears:
5. NEW FILENAME:	Enter the name which MPACK should give to the restructured file. After the MPACK, you end up with the original file (intact) and a restructured version of this file with the filename you specified here. If you enter the name of an existing file, MPACK will return an error message.
6. ERR/LOG FILE?	Specify optional error/log filename for errors and milestone counts: enter a (CR) if no err/log file is desired.
7. MILESTONE COUNT?	Enter appropriate number of records after which a milestone report should be generated. (optional)

The file is then unlocked and restructured, and the user is returned to PRIMOS command level.

Some Examples

The example below shows how a somewhat lopsided version of the CUSTOMER file could be reorganized with MPACK. The file was accessed both by BASIC/VM and PL/I programs, causing a disparity in the number of primary and secondary index entries. There are more entries in the secondary index subfiles than there are records in the data subfile because PL/I does not know how to read (or delete) secondary index subfile entries. By using the ALL or DATA options, the file can be restructured. The first example uses the "ALL" option; the second, the "DATA" option.

Example 1: The ALL Option:

```
OK, creatk
[CREATK rev 17.6]

MINIMUM OPTIONS? yes

FILE NAME? customer
NEW FILE? no

FUNCTION? u

INDEX? Ø

ENTRIES INDEXED:       7
ENTRIES INSERTED:      Ø
ENTRIES DELETED:       2
TOTAL ENTRIES IN FILE:       5

LAST MODIFIED BY MIDAS REV. 17.6

FUNCTION? u

INDEX? 1

ENTRIES INDEXED:       7
ENTRIES INSERTED:      Ø
ENTRIES DELETED:       Ø
TOTAL ENTRIES IN FILE:       7

LAST MODIFIED BY MIDAS REV. 17.6

FUNCTION? u

INDEX? 2

ENTRIES INDEXED:       7
ENTRIES INSERTED:      Ø
ENTRIES DELETED:       Ø
TOTAL ENTRIES IN FILE:       7

OK, mpack
[MPACK rev 17.6]

FILE NAME? customer

'MPACK' OR 'UNLOCK': mpack
ENTER LIST OF INDEXES, 'ALL', OR 'DATA': all
ENTER LOG/ERROR FILE NAME:
ENTER MILESTONE COUNT: 1
```

```
BEGINNING PRIMARY INDEX (INDEX Ø)
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM     DIFF
       Ø  Ø7-21-8Ø  15:47:12     Ø.ØØØ      Ø.ØØØ      Ø.ØØØ     Ø.ØØØ
       1  Ø7-21-8Ø  15:47:14     Ø.ØØ2      Ø.Ø1Ø      Ø.Ø12     Ø.Ø12
       2  Ø7-21-8Ø  15:47:15     Ø.ØØ3      Ø.Ø1Ø      Ø.Ø12     Ø.ØØ1
       3  Ø7-21-8Ø  15:47:15     Ø.ØØ3      Ø.Ø1Ø      Ø.Ø13     Ø.ØØØ
       4  Ø7-21-8Ø  15:47:15     Ø.ØØ4      Ø.Ø1Ø      Ø.Ø13     Ø.ØØØ
       5  Ø7-21-8Ø  15:47:15     Ø.ØØ4      Ø.Ø1Ø      Ø.Ø14     Ø.ØØØ
INDEX Ø MPACK COMPLETE,       5. ENTRIES INDEXED
       5  Ø7-21-8Ø  15:47:17     Ø.ØØ5      Ø.Ø11      Ø.Ø16     Ø.ØØ2

BEGINNING SECONDARY INDEX  1
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM     DIFF
       Ø  Ø7-21-8Ø  15:47:17     Ø.ØØØ      Ø.ØØØ      Ø.ØØØ     Ø.ØØØ
       1  Ø7-21-8Ø  15:47:19     Ø.ØØ1      Ø.ØØ1      Ø.ØØ3     Ø.ØØ3
       2  Ø7-21-8Ø  15:47:21     Ø.ØØ2      Ø.ØØ2      Ø.ØØ5     Ø.ØØ2
       3  Ø7-21-8Ø  15:47:21     Ø.ØØ3      Ø.ØØ2      Ø.ØØ5     Ø.ØØØ
       4  Ø7-21-8Ø  15:47:21     Ø.ØØ3      Ø.ØØ2      Ø.ØØ6     Ø.ØØ1
       5  Ø7-21-8Ø  15:47:21     Ø.ØØ3      Ø.ØØ2      Ø.ØØ6     Ø.ØØØ
       6  Ø7-21-8Ø  15:47:23     Ø.ØØ4      Ø.ØØ2      Ø.ØØ6     Ø.ØØØ
       7  Ø7-21-8Ø  15:47:23     Ø.ØØ4      Ø.ØØ2      Ø.ØØ7     Ø.ØØ1
INDEX  1 MPACK COMPLETE       5. ENTRIES INDEXED
       7  Ø7-21-8Ø  15:47:25     Ø.ØØ6      Ø.ØØ3      Ø.ØØ9     Ø.ØØ2

BEGINNING SECONDARY INDEX  2
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM     DIFF
       Ø  Ø7-21-8Ø  15:47:25     Ø.ØØØ      Ø.ØØØ      Ø.ØØØ     Ø.ØØØ
       1  Ø7-21-8Ø  15:47:27     Ø.ØØ1      Ø.ØØ1      Ø.ØØ3     Ø.ØØ3
       2  Ø7-21-8Ø  15:47:27     Ø.ØØ2      Ø.ØØ1      Ø.ØØ3     Ø.ØØØ
       3  Ø7-21-8Ø  15:47:27     Ø.ØØ2      Ø.ØØ1      Ø.ØØ3     Ø.ØØØ
       4  Ø7-21-8Ø  15:47:28     Ø.ØØ3      Ø.ØØ2      Ø.ØØ5     Ø.ØØ1
       5  Ø7-21-8Ø  15:47:28     Ø.ØØ3      Ø.ØØ2      Ø.ØØ5     Ø.ØØØ
       6  Ø7-21-8Ø  15:47:28     Ø.ØØ4      Ø.ØØ2      Ø.ØØ5     Ø.ØØØ
       7  Ø7-21-8Ø  15:47:28     Ø.ØØ4      Ø.ØØ2      Ø.ØØ6     Ø.ØØØ
INDEX  2 MPACK COMPLETE       5. ENTRIES INDEXED
       7  Ø7-21-8Ø  15:47:29     Ø.ØØ6      Ø.ØØ2      Ø.ØØ8     Ø.ØØ2
OK, creatk
[CREATK rev 17.6]

MINIMUM OPTIONS? yes

FILE NAME? customer
NEW FILE? no

FUNCTION? u

INDEX? Ø

ENTRIES INDEXED:        5
ENTRIES INSERTED:       Ø
ENTRIES DELETED:        Ø
TOTAL ENTRIES IN FILE:        5
```

```
LAST MODIFIED BY MIDAS REV. 17.6

FUNCTION? u

INDEX? 1

ENTRIES INDEXED:       5
ENTRIES INSERTED:      Ø
ENTRIES DELETED:       Ø
TOTAL ENTRIES IN FILE:       5

LAST MODIFIED BY MIDAS REV. 17.6

FUNCTION? u

INDEX? 2

ENTRIES INDEXED:       5
ENTRIES INSERTED:      Ø
ENTRIES DELETED:       Ø
TOTAL ENTRIES IN FILE:       5

LAST MODIFIED BY MIDAS REV. 17.6

FUNCTION? q
OK,
```

Example 2: The DATA Option:

```
OK, mpack
[MPACK rev 17.6]

FILE NAME? customer

'MPACK' OR 'UNLOCK': m
ENTER LIST OF INDEXES, 'ALL', OR 'DATA': d
OK TO OVERWRITE THE FILE? n
ENTER NEW MIDAS FILE NAME: cust.redone
ENTER LOG/ERROR FILE NAME: out
ENTER MILESTONE COUNT: 1

BEGINNING PRIMARY INDEX (INDEX Ø)
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM      DIFF
       Ø  Ø7-24-8Ø  1Ø:22:Ø7      Ø.ØØØ      Ø.ØØØ      Ø.ØØØ      Ø.ØØØ
       1  Ø7-24-8Ø  1Ø:22:Ø8      Ø.ØØ2      Ø.ØØ9      Ø.Ø11      Ø.Ø11
       2  Ø7-24-8Ø  1Ø:22:Ø8      Ø.ØØ3      Ø.ØØ9      Ø.Ø12      Ø.ØØ1
       3  Ø7-24-8Ø  1Ø:22:Ø8      Ø.ØØ3      Ø.ØØ9      Ø.Ø12      Ø.ØØ1
       4  Ø7-24-8Ø  1Ø:22:Ø8      Ø.ØØ4      Ø.ØØ9      Ø.Ø13      Ø.ØØ1
       5  Ø7-24-8Ø  1Ø:22:Ø9      Ø.ØØ5      Ø.ØØ9      Ø.Ø13      Ø.ØØ1
INDEX Ø MPACK COMPLETE,       5. ENTRIES INDEXED
       5  Ø7-24-8Ø  1Ø:22:Ø9      Ø.ØØ6      Ø.ØØ9      Ø.Ø14      Ø.ØØ1
```

```
BEGINNING SECONDARY INDEX  1
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM     DIFF
       Ø  Ø7-24-8Ø  1Ø:22:Ø9     Ø.ØØØ      Ø.ØØØ      Ø.ØØØ      Ø.ØØØ
       1  Ø7-24-8Ø  1Ø:22:Ø9     Ø.ØØ1      Ø.ØØ1      Ø.ØØ2      Ø.ØØ2
       2  Ø7-24-8Ø  1Ø:22:1Ø     Ø.ØØ2      Ø.ØØ2      Ø.ØØ3      Ø.ØØ1
       3  Ø7-24-8Ø  1Ø:22:1Ø     Ø.ØØ2      Ø.ØØ2      Ø.ØØ4      Ø.ØØ1
       4  Ø7-24-8Ø  1Ø:22:1Ø     Ø.ØØ3      Ø.ØØ2      Ø.ØØ4      Ø.ØØ1
       5  Ø7-24-8Ø  1Ø:22:12     Ø.ØØ3      Ø.ØØ2      Ø.ØØ5      Ø.ØØ1
       6  Ø7-24-8Ø  1Ø:22:12     Ø.ØØ4      Ø.ØØ2      Ø.ØØ6      Ø.ØØ1
INDEX  1 MPACK COMPLETE       5. ENTRIES INDEXED
       6  Ø7-24-8Ø  1Ø:22:13     Ø.ØØ5      Ø.ØØ2      Ø.ØØ7      Ø.ØØ1

BEGINNING SECONDARY INDEX  2
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM     DIFF
       Ø  Ø7-24-8Ø  1Ø:22:13     Ø.ØØØ      Ø.ØØØ      Ø.ØØØ      Ø.ØØØ
       1  Ø7-24-8Ø  1Ø:22:16     Ø.ØØ1      Ø.ØØ2      Ø.ØØ3      Ø.ØØ3
       2  Ø7-24-8Ø  1Ø:22:16     Ø.ØØ2      Ø.ØØ2      Ø.ØØ4      Ø.ØØ1
       3  Ø7-24-8Ø  1Ø:22:16     Ø.ØØ2      Ø.ØØ2      Ø.ØØ4      Ø.ØØØ
       4  Ø7-24-8Ø  1Ø:22:17     Ø.ØØ3      Ø.ØØ2      Ø.ØØ5      Ø.ØØ1
       5  Ø7-24-8Ø  1Ø:22:17     Ø.ØØ4      Ø.ØØ2      Ø.ØØ6      Ø.ØØØ
       6  Ø7-24-8Ø  1Ø:22:18     Ø.ØØ4      Ø.ØØ2      Ø.ØØ6      Ø.ØØ1
INDEX  2 MPACK COMPLETE       5. ENTRIES INDEXED
       6  Ø7-24-8Ø  1Ø:22:19     Ø.ØØ5      Ø.ØØ3      Ø.ØØ8      Ø.ØØ2
OK, creatk
[CREATK rev 17.6]

MINIMUM OPTIONS? yes

FILE NAME? cust.redone
NEW FILE? no

FUNCTION? u

INDEX? Ø

ENTRIES INDEXED:        5
ENTRIES INSERTED:        Ø
ENTRIES DELETED:        Ø
TOTAL ENTRIES IN FILE:        5

LAST MODIFIED BY MIDAS REV. 17.6

FUNCTION? u

INDEX? 1

ENTRIES INDEXED:        5
ENTRIES INSERTED:        Ø
ENTRIES DELETED:        Ø
TOTAL ENTRIES IN FILE:        5

LAST MODIFIED BY MIDAS REV. 17.6
```

MPACK ERROR MESSAGES

In addition to the messages returned through it by KX$RFC (those messages are listed in Appendix A), MPACK returns the following error messages. A "fatal" error means that MPACK cannot continue processing as a result of the error, and will promptly abort. A "non-fatal" error is a warning only and does not impair the MPACK process.

▶ UNABLE TO REACH BOTTOM INDEX LEVEL

MPACK was unable to find a last level index block for an index: the file is damaged. (Fatal)

▶ INDEX SUBFILE DOES NOT EXIST

User supplied an index number that was not defined for this file. (Non-fatal)

▶ FILE ALREADY EXISTS -- TRY AGAIN

User specified name of existing file in response to "ENTER NEW MIDAS FILE NAME?" prompt of "DATA" option path. MPACK will not overwrite an existing file in this case; the user must enter the name of a non-existent file. (Non-fatal)

▶ INVALID KEY SEEN (IGNORED)

This can happen when a key is out of order in the index or the key is a duplicate and duplicates are not allowed in the primary index. (Non-fatal)

▶ INVALID DIRECT ACCESS ENTRY NUMBER SEEN (IGNORED)

Can occur when a record number is not greater than zero, or is not a whole number, or is greater than the pre-allocated record number limit. (Non-fatal)

▶ DATA SUBFILE FULL

This may occur if MPACK is being used to implement smaller index blocks, smaller segment subfiles or smaller segment directories. (Fatal)

▶ INDEX FULL

Same as above, only an index subfile is suffering in this case. (Fatal)

▶ ABORTING MPACK

Message appears when a "fatal" error occurred. The user should then delete the MPACK-created scratch files by using the "Junk" option of KIDDEL; or, if doing a "DATA" MPACK, delete the "new file."

SECTION 13

FOR THE ADMINISTRATOR

INTRODUCTION

This section is intended primarily for the MIDAS Administrator, or anyone else who is involved in setting up MIDAS and monitoring its daily use. It addresses the following fundamental concerns:

- Concurrent process handling
- Setting MIDAS file read/write locks
- Using MIDAS with networks
- Initializing MIDAS with IMIDAS
- Library modifications
- The new V-mode interlude
- Releasing the internal lock using MCLUP
- Recovering/restructuring MIDAS file entries using MPACK
- Concurrency error detection
- Debugging tips

Important!

Many of the tasks involved in MIDAS administration are directly related to the present method of concurrent process handling. This affects all new users of MIDAS, users of existing applications, and particularly network users, making it imperative that you familiarize yourself with the information in this section before bringing up the latest version of MIDAS on your system.

CONCURRENT PROCESS HANDLING

Overview

Concurrent process handling refers to MIDAS's method of regulating simultaneous access to a MIDAS file by two or more processes. Ideally, such a method should not impair program performance and should prevent conflicting updates.

Implementation Method

The concurrent process handling mechanism is implemented through the use of several constructs:

- MIDAS file read/write locks are set at 3; n readers and m writers.
- A "lock" in shared memory that keeps track of who's currently executing a MIDAS operation
- A semaphore wait list - a queue of processes waiting to perform MIDAS operations
- The MIDAS file open and close routines, OPENM$ and CLOSM$ (see Section 4), and the "notify MIDAS" routine, NTFYM$, which tells MIDAS when a file is opened and closed (a substitute for OPENM$ and CLOSM$; used with PRIMOS SRCH$$ routine)

Setting File Read/Write Locks

MIDAS no longer closes segment subfiles (index subfiles) between MIDAS operations. This requires that users set MIDAS file read/write locks to 3 (n readers and m writers); otherwise, concurrent processes will be unable to open a file segment which had already been opened by another process. This allows more than one person to have the file open for writing. CREATK automatically sets the MIDAS file read/write lock to 3 whenever it creates a new file. However, you should take care to set the locks on existing MIDAS files to 3 when bringing up the latest revision of MIDAS.

Note

FUTIL automatically sets the read/write locks to the system default when it copies a MIDAS file from one place to another; reset the lock by typing:

```
SR filename 3
```

where filename is the name of the copied MIDAS file.

The Shared Lock

To prevent the loss of file integrity by conflicting updates which could conceivably occur with read/write locks set at 3, MIDAS employs a "lock-testing" procedure to allow only one user to execute a MIDAS operation at a time. This is known as "single-threading".

Every time a call is made to MIDAS, a shared lock, located in shared memory segment 2020, is tested. If the lock value is zero when tested, the lock is set to the testing process's user number and the testing process is allowed to perform a MIDAS operation. A process can only obtain the lock if the lock is set at zero when the process checks it. Otherwise, some other process "owns" the lock, forcing the testing process to wait until the lock is freed.

Each time a process completes a MIDAS operation, the lock is released and the value is reset to zero. During the lock-test operation, no other process is allowed to access the lock, making the "test-and-set" operation non-interruptible. This eliminates the problem of having a process alter the lock value while another process is testing it.

The Semaphore Wait List

Should the lock have a non-zero value when tested, some other process "owns" MIDAS and the testing process must wait its turn on a "semaphore wait list". The wait list is just a queue of processes waiting in an idle state to obtain exclusive access to MIDAS so they can perform their respective operations. Only one process can access MIDAS at a time, thus preventing damage to file integrity by simultaneous updates to the same file record. While more than one process may have the same MIDAS file open at a time, only one of these processes can actually operate on the file at any given time because only the process that owns the lock can actually touch the file. When a process releases the MIDAS lock, it performs a "notify" operation on the semaphore, telling it to activate the first process waiting on the queue (it's a FIFO queue); the next user is then released from the wait list.

What This Means to New Users

This concurrency handling method is entirely invisible to new users; however, note that OPENM$ and CLOSM$ or NTFYM$ should be used in opening and closing a MIDAS file in order for the concurrency handler to operate properly. These routines are documented in Section 6; they are handled automatically by COBOL and RPG. Network users, however, cannot use this method of handling concurrent processes. (See below for details.)

Impact on Existing Applications

Users with existing MIDAS application programs written prior to Rev. 16.5 have two choices:

1. Disable the concurrent process handling method; no changes to existing programs are necessary. This would result in no performance optimization, but detection and correction of currency errors will still occur. Follow the steps under "Disabling Concurrent Process Handling", below. PRIMENET users: see Note, below.

2. Some or all applications programs can be modified, achieving increased performance. Follow the directions for program modification, listed below.

Note

PRIMENET users should disable the concurrent process handling method by following the steps listed below. If this is not done, MIDAS will not single-thread MIDAS use or correct concurrency errors for files accessed over the net. When the current method of concurrent process handling is disabled, MIDAS defaults to the "old" method of concurrency handling. See OLD METHODS in Appendix D.

Modifying Existing Programs

If you want to use the new method of handling concurrent processes, MIDAS must be notified both when a process is to begin using a MIDAS file and when the process has completed operations on the file. For FORTRAN and PMA users of the MIDAS call level interface, this requirement means that application programs must be modified in one of two ways, described below. COBOL, BASIC/VM, PL/I and RPGII users are not required to make any modifications to existing or new applications programs written in these languages. All programs which use an unshared library must be reloaded. See also LIBRARY MODIFICATIONS and The V-Mode Interlude.

Method 1: Use NTFYM$: One method of program modification involves inserting calls to subroutine NTFYM$. The first call to NTFYM$ should be inserted following the call to open the MIDAS file and just prior to the first MIDAS file operation. The other call to NTFYM$ should be inserted just before the call to close the MIDAS file. This is described in detail in Section 6. When using NTFYM$, a MIDAS file can be opened and closed by general methods like the SRCH$$ routine.

Method 2: Use OPENM$ and CLOSM$: The second method is to replace the calls which open and close MIDAS files with calls to OPENM$ and CLOSM$ respectively. Details are provided in Section 6.

Disabling Concurrent Process Handling

Users may disable the new concurrent process handler, thereby returning to the method used in previous releases of MIDAS. Note that programs which use the new NTFYM$, OPENM$, and CLOSM$ routines will still work correctly, even if this process is disabled. However, performance degradation may result.

On page 13-5, there should be a step 4 indicating that the read/write file locks on all MIDAS files should be changed to 2 (n readers and 1 writer) when disabling concurrent process handling.

The following steps should be followed in disabling concurrent process handling:

1. In file KPARAM, change the value of parameter SHDSEG from .TRUE. to .FALSE..

2. For the unshared MIDAS library NVKDALB, run the command file C_NVKDALB and C_INSTALLMIDAS and reload applications programs which use the unshared libraries.

3. For the shared V-mode library VKDALB, rebuild the library using C_VKDALB and re-install MIDAS using C_INSTALLMIDAS and C_SHAREMIDAS. Application programs which use the shared library do not need to be reloaded.

4. *see PTU89*

INITIALIZING MIDAS

MIDAS installation information is included with the release of the master disk at each major revision. Directions for installation are covered there and are not dealt with in this book. A summary of what the new command files do and where they're located is included below for quick reference, along with a list of the sub-UFDs contained in the MIDAS UFD:

Sub-UFD's in MIDAS UFD

CMDNCØ	Contains MIDAS utilities CREATK, KBUILD, KIDDEL, IMIDAS, MCLUP, REVERT, REMAKE, and MPACK.
LIB	Contains MIDAS libraries VKDALB, NVKDALB and KIDALB.
SOURCE	Contains all source and command files: see below for details.
SYSTEM	Contains files K4ØØØ, K2Ø14A and K2Ø14B.

Command Files in MIDAS

C_MIDAS	Builds MIDAS by calling C_KIDALB, C_NVKDALB, C_VKDALB, C_IMIDAS, C_MCLUP, C_CREATK, C_KIDDEL, C_KBUILD, and C_MPACK. Copies PARM.K and PARM.K.PLI (PL/I insert file) from MIDAS>SOURCE to MIDAS>SYSCOM.
C_INSTALLMIDAS	Installs MIDAS by copying various MIDAS command files and utilities to system UFD's SYSTEM, SYSCOM, LIB and CMDNCØ.
C_SHAREMIDAS	Shares MIDAS library, shared lock segment and runs SYSTEM>IMIDAS to intialize all of MIDAS. This file can only be run from the system console.

Command Files in MIDAS>SOURCE

C_CREATK	Builds CREATK in MIDAS>CMDNCØ.
C_IMIDAS	Builds utility IMIDAS in UFD MIDAS>SYSTEM.
C_KBUILD	Builds KBUILD in MIDAS>CMDNCØ.
C_KIDALB	Builds the R-mode library in UFD MIDAS>LIB.
C_KIDDEL	Builds KIDDEL in MIDAS>CMDNCØ.
C_LCREATK	Builds long index version of CREATK; see Section 15.
C_LIST_MIDAS	Makes a compilation listing of the major MIDAS routines.
C_MCLUP	Builds utility MCLUP in UFD MIDAS>CMDNCØ.
C_MIDAS	Builds MIDAS libraries and utilities.
C_MPACK	Builds MPACK in MIDAS>CMDNCØ. (See Section 12 for information on options.)
C_NVKDALB	Builds the unshared V-mode library NVKDALB in UFD MIDAS>LIB.
C_REMAKE	Builds obsolete utility REMAKE in MIDAS>CMDNCØ. See Appendix C.
C_REVERT	Builds obsolete utility REVERT in MIDAS>CMDNCO. See Appendix C.
C_VKDALB	Builds the shared V-mode library, VKDALB. VKDALB is put in MIDAS>LIB. K4ØØØ, K2Ø14A, and K2Ø14B are placed in UFD MIDAS>SYSTEM.

IMIDAS: Initialization Utility

The present method of concurrent process handling requires a special initialization procedure to be run during each and every cold start.

This must be run to share the segment containing the shared lock, and to set the lock value to zero. Make sure that no MIDAS applications programs are running at the time IMIDAS is invoked. You may want to install the command file C_SHAREMIDAS in the cold start procedure to ensure that IMIDAS is executed and that the lock segment is shared during every cold start.

Lock Parameters: Currently, MIDAS uses semaphore number -16 and word number '177777 of segment 2Ø2Ø (the lock). These parameters, defined in file KPARAM, may be modified:

MSEMA1 semaphore number

SLSEG segment number of the shared lock

SLWORD word number of the shared lock

If any of these parameters are modified, the MIDAS utilities MCLUP (see below) and IMIDAS must also be rebuilt and installed. In addition, command file C_SHAREMIDAS must be modified so that the correct segment gets shared. Remember that the MIDAS libraries, MCLUP and IMIDAS must be rebuilt also. See Section 15 for information on modifying these and other MIDAS parameters.

Note

See also MCLUP, under MIDAS CLEANUP/RECOVERY UTILITIES, below.

LIBRARY MODIFICATIONS

There have been several changes to the MIDAS libraries at this rev, resulting in greater flexibility for both R-mode and V-mode users, and in overall program performance improvement.

The Interlude

At this rev of MIDAS, the R-mode MIDAS library, KIDALB, and the R-mode runtime routines have been replaced by an interlude to the V-mode library, VKDALB. This change results in several positive benefits, including:

- reduced memory requirements -- only one copy of the shared V-mode library, VKDALB, is needed instead of one copy of the R-mode routines per user.
- reduced I/O (CPU) time -- V-mode library uses the PCL instruction which is much quicker than the SVC instruction used by the R-mode library to implement a file system call. Since a single MIDAS call may result in many file system calls, the implications are obvious.
- only one library to worry about
- R-mode programs using only the MIDAS library don't need to be reloaded. However, R-mode programs using other R-mode routines only need re-loading once. The new shared library is dynamically linked during each execution, it is only necessary to reload KIDALB initially when the new version of MIDAS is installed.

Note

This interlude is not available to users of the PRIME P-300.

Impact of Interlude

The new interlude brings with it a few miscellaneous changes in the availability of certain internal and offline routines. The only internal MIDAS routines callable by R-mode and V-mode users are: ERROPN, FILERR, FILHER and KX$TIM. Also available are the offline routines PRIBLD, SECBLD and BILD$R.

Previously, V-mode users were not granted access to any internal routines at all, so this represents an enhancement to V-mode users. To R-mode users, this represents a restriction, as they used to have access to all internal routines. Since the R-mode library essentially tracks the V-mode library in functionality, only these four routines just mentioned will be available. Users who've included calls to these routines in the past should note that the calling sequences have changed; see Appendix D for more information.

MIDAS CLEANUP/RECOVERY UTILITIES

MIDAS provides two utilities for cleanup and recovery: one for reinitializing the shared lock after an abnormal program termination and one to recover or restructure deleted index entries in a MIDAS file.

MCLUP: Releasing the Internal Lock

In the event of abnormal MIDAS program termination, MCLUP must be used to re-initialize the shared lock and to notify the semaphore to awaken any MIDAS process waiting on the lock. If the lock is not released, all MIDAS processes will be suspended indefinitely. It's quite possible that such a condition might go unnoticed for quite some time. Should this condition be suspected, use MCLUP to determine who "owns" the lock.

The MCLUP command takes the following form:

```
MCLUP [-USER userno]
```

If used without options, MCLUP re-initializes the shared lock only if it is held by the user who issued the command. If this user is not the "owner" of the lock, MCLUP prints the user number of the user that does own it.

When used with the "-USER userno" option, MCLUP initializes the lock only if held by the indicated user number; otherwise, it prints the number of the user that holds the lock. Simply reissue MCLUP with the

proper user number to release the lock so other processes can proceed.

Automatic Notify: Sometimes when a user quits out of a process or logs off the system, the lock is released but the subsequent "notify" to the MIDAS semaphore does not have a chance to happen and MIDAS will cease to function. The MCLUP utility should then be invoked in its optionless form. In this circumstance, MCLUP will generate an automatic notify to the semaphore as long as there is at least one user on the wait-list, and nobody else has "grabbed" the lock. This "automatic notify" feature prevents MIDAS from hanging up indefinitely because it can't signal another waiting process to ask for the LOCK. MCLUP prints out this message at the terminal of the user who invokde MCLUP whenever an automatic notify successfully generated:

```
Cleanup for unknown user successful
```

HANDLING CONCURRENCY ERRORS

Although most concurrency errors can be avoided through the lock method described above, there are some that single-threading cannot prevent. MIDAS is able to detect and correct most concurrency errors, which may occur when more than one user accesses the same MIDAS file at the same time. These errors are usually associated with operations that involve the current record (the record just read, written or updated), and occur when the current index entry has been deleted or physically moved since the time the entry became current. If MIDAS discovers that the entry has been deleted, an error code of 13 is returned. In the event that the entry has been moved, MIDAS automatically locates the entry and continues normally.

How MIDAS Traces the Current Record

At the FORTRAN call level interface, the concept of current record and current entry is implemented as a 14-word communication array. The communication array is an argument in most subroutine calls to MIDAS. The communications array format is described below. Other information on this array can be found in Section 6.

Communications Array Format

The fourteen words of the array contain the following information:

Word 1: (input) if Ø or 1, the array contents are used
if -1 then MIDAS array contents are not used
(output) error status

Word 2-4: current index entry address

word 2 bits 1-8	entry number
word 2 bits 9-16	segment file number
words 3 - 4 (32 bits)	word offset of index block

Word 5: hash value (based on current key value)
(for keys longer than 8 bytes (4 words)

Words 6-9: current key value (or first four words of key)

Words 1Ø-12: current record address

word 1Ø bit 1	"record locked" flag
word 1Ø bits 7-16	segment file number
words 11 - 12	word offset of record

Word 13: data control word

bits 1-8	flag bits
bits 9-16	primary key size (bits)

Word 14: data record length (in words)

Note that words 2 through 9 of the communication array specify a current index entry and words 1Ø through 12 specify a current record.

How MIDAS Uses the Array: During operations involving the current entry (for example, "get next record"), words 2 through 4 are used to locate the expected position of the entry. To verify that the position contains the correct entry, MIDAS compares the data pointer in the entry with the data pointer in words 1Ø through 12 of the communication array. If the pointers don't match, then the entry is the wrong one.

Even if the pointers do match, MIDAS compares the key value in the index entry to the key value in the array. If they don't match, then the entry is the wrong one. When a wrong entry is detected, MIDAS searches for the correct entry. If the correct entry is not found, MIDAS returns an error code of 13. Note that versions of MIDAS between Rev 16.Ø and Rev 16.5 returned an error code of 13 when a concurrency error was detected. Users of these earlier releases may have modified their applications to attempt to recover from an error 13. Since an error 13 presently indicates that the current index entry has been deleted, attempts to handle an error 13 in existing applications may

have to be modified.

Limitations on Error Detection

For indexes with keys longer than eight bytes, MIDAS may fail to detect a concurrency error or to correctly recover from one. To understand how this may occur, observe that a maximum of eight bytes of a key may be stored in the communication array. For keys longer than eight bytes, MIDAS stores a hash value in word 5 of the array. The hash value is based on the portion of the key beyond the eighth byte. MIDAS will fail to detect a concurrency error only if:

- The data pointers match and the first eight bytes of the key match the eight bytes stored in the communication array

  or

- The data pointers match and the hash code, based on the remaining bytes, is the same as the hash code in the array

DEBUGGING TIPS

The MIDAS administrator will doubtless be confronted with occasional MIDAS-related problems. Before attempting to tackle them, it is suggested that you take time to check the following points.

MIDAS Revision Number

Do you have all the documentation pertaining to the particular revision of MIDAS you're using? Use the VERSION option of CREATK to determine what software revision of MIDAS you're running and which version the file was created under.

Incompatible Interfaces

Does the application program in question use both a language interface like COBOL and direct calls to MIDAS through FORTRAN/PMA subroutines in the same program? This can lead to confusion on the part of the user and ultimately to a damaged file.

Restructure History

When was MPACK last run against the file in question? With the USAGE option of CREATK, check each index and determine the percentage of the file that has been deleted. Performance degradation can result when there are lots of deleted entries in secondary indexes that contain many duplicates. Use MPACK to unlock records; see Section 12.

Hint: always use the "NEW" option of MPACK to prevent possible damage to the original file in case of error or program failure.

Check the Locks

The latest version of MIDAS has two locks with which users must be concerned; the per file read/write lock and the shared internal lock.

Read/Write Locks: Check to make sure that per file read/write locks on MIDAS files have been set to 3. Prior to Rev. 16.5, MIDAS read/write locks were set at 2 (the standard system-wide setting), but must be set at 3 to permit the new method of concurrent process handling to operate. Note that the lock setting is not preserved by MAGSAV or FUTIL TRECPY, so check the lock each time a file is processed by either of these operations. See Setting File Read/Write Locks, above.

Internal Lock: The shared internal lock, used in gating concurrent processes, must be initialized with IMIDAS as described under INITIALIZING MIDAS, earlier in this section. The latest version of MIDAS specifies the shared lock to be word number '177777 of segment 2020. The lock segment is not protected from being accidentally overwritten by any user, so be careful!

New Calls and Libraries

Are all FORTRAN/PMA programmers using the new OPENM$ and CLOSM$ or NTFYM$ calls? If not (and intentionally), has the concurrent process handler been disabled correctly as described earlier? If not, performance may be noticeably degraded. Have all other language interface programs been reloaded with the new libraries?

Check the Semaphore

The semaphore for the concurrent process wait list has gone through a series of values since its debut. Currently, the semaphore value is -16; this value works only on MIDAS versions 17 and above. Earlier versions of MIDAS used a setting of 64, which works for both Rev 16 and 17 versions of MIDAS.

Are you using Networks?

Remember that network users cannot operate with the shared lock and semaphore-based concurrent process handler; either make sure that the this has been disabled according to the instructions outlined earlier in this section, or set the MIDAS file locks to 2. This causes the old method of concurrent process handling to be used. See Appendix D for a description. Network users may still use the new OPENM$, CLOSM$ and NTFYM$ calls in their programs, as these routines are not affected by disabling the lock-testing procedure.

Note

For systems using earlier versions of MIDAS and/or MIDAS files created under previous versions, see also Appendix D.

Multiple Libraries

Network users may want to make a separate copy of the MIDAS library for their own use. This would be an unshared library and would contain all the modifications indicated previously under Disabling Concurrent Process Handling and IMIDAS: Initialization Utility, above. Making all the necessary changes once and saving them in a separate copy of the library which can be used whenever needed is a lot easier than making the changes every time you want to use MIDAS across the net.

Part V
More Midas

SECTION 14

OFFLINE ROUTINES

INTRODUCTION

The current version of MIDAS makes many offline routines available for general use. Most of these routines, including the file-building routines and error-logging routines have been used by MIDAS prior to this revision, and were previously made available to R-mode MIDAS users only. In conjunction with the file-building routines, a few newly-added routines make it possible for the user to design, create, and populate a MIDAS file without using the MIDAS utilities CREATK or KBUILD.

What's In This Section

To simplify matters, this section is divided into three parts: the first part shows how to create and look at a MIDAS file template using the KX$CRE and KX$RFC routines; the second part deals with the offline file-building routines, PRIBLD, SECBLD, and BILD$R, and the third part covers miscellaneous routines like ERROPN and KX$TIM, some of which may be familiar to users of previous versions of MIDAS.

PART I. CREATING/EXAMINING A MIDAS FILE

ALTERNATIVES TO CREATK

Available for the first time at this revision of MIDAS are the user-callable internal routines KX$CRE and KX$RFC. KX$CRE is the "work horse" of the CREATK utility, while KX$RFC is used by nearly every MIDAS utility to return the configuration of a MIDAS file. These two routines can be used in place of CREATK as an alternate method of building and looking at the configuration of a MIDAS file.

Users will find this material useful only if they want to write their own programs to create file templates and read file configurations instead of using command files that invoke CREATK and/or KBUILD. One advantage of programs over command files is that they can handle errors neatly and are not subject to subtle changes made in utility interfaces.

KX$CRE

KX$CRE is a user-callable routine that can be used to create a MIDAS file from a program. It is the same routine used by CREATK to create new MIDAS files and exists in all the MIDAS libraries.

KX$CRE Calling Sequence

The calling sequence of KX$CRE is:

```
CALL KX$CRE (filnam,namlen,flags,alloc,pridef,secdef,errcod)
```

The arguments used in the above call and their data types (in parentheses) are:

filnam — The pathname of the file to be opened, two characters per word (INT*2).

namlen — Length of filnam in characters (INT*2). DEFALT

flags — Global flags (INT*2): see The Flags Argument, below.

alloc — Number of data records to pre-allocate if direct access is enabled for this file (REAL*4). Specify this number only if M$DACC is set in flags. This argument is ignored when creating a keyed-index file and may be set at Ø.

pridef(6) — Definition array for the primary index (INT*2). See Table 14-1.

secdef(6,17) — Definition array for the 17 secondary indexes (INT*2). secdef(1...6,i) contains the definition for secondary index "i," where i ranges from 1 to 17. See Table 14-1.

errcod(2) — Error code returned by MIDAS (INT*2). If the error code in errcod(1) is Ø, it indicates successful completion. If the code is less than 5ØØØ, then it is a file system error and KX$CRE tries to delete the partially created file, ignoring any errors incurred in the process. If errcod(1) is 5ØØØ or greater, then the error code relates an error in the MIDAS file definition passed by the user, errcod(2) containing an index number, if applicable. See Non-File System Error Codes below.

On page 14-2, the argument namlen does not have to be declared as INT*2, as implied in the book. This is the default however.

The Flags Argument: The flags argument indicates the file type and the READ/WRITE lock setting of the file to be created in this call.

The flags argument value is set with the following keys:

Key	Function
M$DACC	Enables file for direct access. alloc contains the initial number of records to pre-allocate.
M$NRNW	Sets file lock to n readers and n writers. This is the default even if M$NRNW is not specified.
M$NR1W	Sets file lock to n readers and 1 writer. This must be used if the file is intended for access by more than one user at a time and at least one of those users is to access the file over PRIMENET.

Note

The file lock keys, M$NRNW and M$NR1W, are mutually exclusive.

The Pridef and Secdef Arrays: The pridef and secdef arrays must be assigned values (by the user) to indicate the characteristics of the primary index and any secondary indexes to be included in the file template. These arrays are passed by the user's program to KX$CRE, which then uses this information to build the file template.

The six-element one-dimensional array pridef(1...6), contains the information needed to define the primary index. Similarly, secondary indexes are defined by the two-dimensional secdef array, where secdef(1...6,i) defines secondary index "i." All the elements in these arrays are INTEGER*2.

The six elements of each array are listed in Table 14-1. The flags listed in Table 14-2 are divided into three groups. The first group defines special index subfile characteristics; the second defines key type and the third tells whether the key size is supplied in bits, bytes or words. These flags are passed to KX$CRE in much the same way as the MIDAS flags are passed to the various MIDAS subroutines as discussed in Section 6.

Table 14-1. Pridef and Secdef Array Elements

Array Element	Description
pridef(1)	Contains one or more flag values specifying the key type and key size, plus whether or not the primary index is to be double length. See Table 14-2.
secdef(1,i)	Defines the key type and size of the "ith" secondary key: also determines the duplicate status of the key and defines the length of the index subfile (i.e., tells if it's double-length). See Table 14-2 for key-type flags.
pridef(2)	Primary key size in bits, bytes or words, depending on key size flag specified in pridef(1).
secdef(2,i)	Secondary key size in bits, bytes or words, as indicated by key size flag supplied in secdef(1,i). If this element is Ø, the index does not exist.
pridef(3)	Data record size. If Ø, means variable-length records.
secdef(3,i)	Secondary data size: see Appendix C. Supply a Ø if this feature is not desired.
pridef(4)	Level 1 block size. Supply a Ø to use default block size (1Ø24 words).
secdef(4,i)	Level 1 block size: same as above.
pridef(5)	Level 2 block size. Supply a Ø to use default block size.
secdef(5,i)	Level 2 block size: same as above.
pridef(6)	Last level block size: Ø = use default.
secdef(6,i)	Same as above.

Table 14-2. Flags for pridef(1) and secdef(1)

Index-Specific Flags

Flag	Meaning
M$DLNG	Double length index
M$DUPP	Duplicates permitted for this key (illegal for the primary key)

Key Type Flags

Flag	Meaning
M$BSTR	Bit String
M$SPFP	Single-precision floating point (REAL*4)
M$DPFP	Double-precision floating point (REAL*8)
M$SINT	Short (one-word) integer (INT*2)
M$LINT	Long (two-word) integer (INT*4)
M$ASTR	ASCII string

Key Size Flags

Flag	Meaning
M$BIT	Key length is specified in bits
M$BYTE	Key length is specified in bytes
M$WORD	Key length is specified in words

Using the Flags: To use the flags, select the appropriate flags, one from each of the three groups, except for the first group from which you may choose none, one or both flags. Assign them to the first word of the pridef or secdef array in the following manner:

```
SECDEF(1,1) = M$DUPP + M$ASTR + M$WORD
```

This defines an ASCII key that allows duplicates, and whose length, defined in words, is supplied in SECDEF(2,1).

Non-File System Error Codes

Errors encountered during the building of a template may originate in the file system or in MIDAS, and may result from bad user arguments or from some internal MIDAS problem. The error codes which the user is most likely to see are listed in Table 14-3.

KX$RFC

KX$RFC is a user-callable routine that returns the file configuration of an already existing MIDAS file. Its calling sequence is essentially the same as that of KX$CRE, except that most of the arguments are returned by KX$RFC instead of being set by the user. Its calling sequence and arguments are described below.

Table 14-3. KX$CRE Error Codes

Code	Meaning
ME$BAS	Allocation size is bad. The number specified in alloc was: (a) less than 1.Ø, (b) not a whole number, or (c) too big to allocate alloc records (the user-supplied number passed in the alloc argument) given the default segment directory and segment subfile lengths.
ME$BDS	Data size is bad for one of these reasons: • Data size is negative. • Data size in pridef(3) is Ø meaning variable-length data records, but the file is configured for direct access which requires fixed-length data records.
ME$BKS	Key size is bad. For example: • Key size is too big. The limit is 16 words except for ASCII strings which may be up to 32 words. • Key size is negative. • Primary key size is Ø.
ME$BKT	Key type is bad.
ME$BL1	Level 1 block size is bad. The block size must be positive, not larger than RECLNT (1Ø24 words), and must hold at least two index entries.

Table 14-3. Error Codes (continued)

ME$BL2	Level 2 block size is bad.
ME$BLL	Last level block size is bad. When building a secondary index, this error may also be caused when the secondary data size, secdef(3,i), is too large, in comparison to the block size, to fit the mandatory two entries per block.
ME$CBD	Index can't be double length. The user may have specified the primary index as double length and then defined secondary index one, or specified secondary index i as double length and then defined secondary index i+1, or the user may have specified secondary index 17 as double length.
ME$NDA	No duplicates allowed. The user specified the flag M$DUPP in pridef(1) that is not permissible because duplicates are not allowed for the primary key.

Note

All of the flags and codes listed above are defined in SYSCOM>PARM.K.

KX$RFC Calling Sequence

The KX$RFC calling sequence is:

```
CALL KX$RFC (filnam,namlen,flags,alloc,pridef,secdef,errcod)
```

The arguments, their meanings and their data types are:

filnam	The pathname of the MIDAS file whose configuration is to be returned (INT*2). (supplied by user)
namlen	Length of filnam in characters (INT*2). (supplied by user)
flags	Global flags: M$DACC is the only possible flag returned (INT*2). (returned by KX$RFC)
alloc	Number of data records pre-allocated if direct access enabled, otherwise Ø.Ø is returned (REAL*4). (returned by KX$RFC)
pridef(6)	Definition array for the primary index (INT*2). (returned by KX$RFC)
secdef(6,17)	Definition array for the 17 secondary indexes (INT*2). secdef(1...6,i) contains the definition for secondary index i. (returned by KX$RFC)
errcod	Error code or Ø if no error (INT*2). (returned by KX$RFC)

The arguments are very similar to those used in calls to KX$CRE, and, with the exception of a few minor notes which appear below, the user is referred to the previous discussion on KX$CRE for details on the arguments used in this call.

Notes on KX$RFC Arguments

The flags argument is not supplied by the user in a call to KX$RFC, but is instead returned by a successful call to that subroutine. If the file is a direct access file, the flag M$DACC is returned; otherwise, flags is returned as Ø. If the file is not enabled for direct access, the alloc argument will be zeroed by KX$RFC.

Pridef and Secdef Flags: The flags returned on this call are:

- M$BSTR,M$MSPFP, M$DPDP, M$SINT, M$LINT and M$ASTR represent possible settings for bits 1-4.
- M$DLNG and M$DUPP represent the setting of bits 5 and 6 respectively.
- MBIT, MBYTE, M$WORD represent possible values of bits 6 - 7. (M$WORD is generally not returned by KX$RFC.)

Element secdef(2,i) is returned as Ø if there is no index "i" in this file.

Errcod: This is a one-word argument in this call instead of a two-word array. Error codes less than 5ØØØ indicate file system errors. The only returned code greater than 5ØØØ is:

ME$NMF

which means this file is not a MIDAS file. This could be true for several reasons, including:

- The file is not a SAM segment directory
- Segment subfile Ø (file descriptor subfile) does not exist

or

- Segment subfile Ø does not contain the appropriate flags to indicate that the file is a MIDAS file

There are other messages returned by KX$RFC. They are listed in Appendix A along with the other MIDAS error messages that are common to several MIDAS routines. KX$RFC is called by nearly all of the MIDAS utilities, and therefore, any message it returns is more than likely to be associated in the user's mind with the routine that was just called when the error occurred. Thus, these messages are treated in the place where the user is most likely to look for explanations of error messages.

On page 14-11, the statement under Why Use Offline Routines? states that off-line routines are faster because they are not shared. This is only part of the actual story. Offline routines are not meant to be shared and therefore are not concerned with multi-user access to a file. Therefore they don't write out index blocks after each index entry is added to a file, as ADD1$ does. (Online routines must always write index blocks out to the file after each operation on the block so that the file will not be damaged by concurrent access and so each user will have a consistent view of the file while accessing it.) By not writing out the index blocks each time, a considerable amount of I/O overhead is saved, making off-line routines faster than their online counterparts. In addition, the off-line routines bypass the concurrent process handling method which normally single-threads MIDAS use for online routines. Thus only one person can have access to a MIDAS file at a time when that file is being processed by an off-line routine.

PART II. FILE-BUILDING ROUTINES

ALTERNATE FILE-BUILDING METHODS

The second part of this section describes several ways of building MIDAS files which were not covered in Section 3. Specifically, there are three offline file-building routines available in the current MIDAS library: PRIBLD, SECBLD and BILD$R. These routines can be called from any user program to add index and data subfile entries to keyed-index or direct access MIDAS files. Data entries can have fixed or variable-length records and can be sorted or unsorted. R-mode users may already be familiar with these routines -- they were previously kept only in the R-mode library. People who've used these routines in the past should consult Appendix D for an explanation of changes made to them with the current version of MIDAS.

Briefly, the functions of these routines are:

- BILD$R builds MIDAS data and index subfiles from sorted/unsorted input data.
- PRIBLD builds an empty primary index subfile and data subfile from sorted input data.
- SECBLD builds an empty secondary index subfile from sorted input data.

You can use these routines by calling them from programs written in FTN, BASIC/VM, PL/I, or any other Prime language. Consult the appropriate reference guides and user manuals for details on calling external routines. However, it is assumed that the offline routines described in this section are most useful to FORTRAN or PMA programmers.

Why Use Offline Routines?

Because offline routines are not shared, only one user can "have" them at a time; thus they are faster than the online routines like ADD1$. If you have networks, you might use PRIBLD, SECBLD and BILD$R as part of your "network library" package; see Section 13 for details.

These routines also provide the user with tools for building files with concatenated keys or adding secondary data entries, neither of which are supported by KBUILD.

see PTU89

Restrictions

Offline routines cannot be called by more than one user to operate on the same file. Once the file is opened by one of these routines, no one else can invoke any MIDAS routine to work on that file. However, it is possible for more than one of these routines to be called from the same program to work on the same file, as long as they don't access the same index subfile concurrently. This allows the user to build an entire file from a single program by calling the file-building routines in the proper sequence.

It is also not possible to build more than one MIDAS file at a time from the same program using offline routines. In addition, it is not a good idea to access a file being built by one or more offline routines or a utility with any of the online routines. Doing so can damage the file.

The offline routines do check to see that they are not accessing the same index subfile of a given MIDAS file simultaneously while building it. However, they cannot guarantee that the file is not being accessed by an online routine or utility while it's being built.

Which Routine to Use

Review these guidelines before deciding which routines to use for MIDAS file-building. Make sure you understand which routines should be used together when building a file.

Note

When using any of the routines discussed in this section, remember to $INSERT all of the following insert files into your FORTRAN programs:

```
SYSCOM>PARM.K
SYSCOM>ERRD.F
SYSCOM>KEYS.F
```

Use PRIBLD: Use PRIBLD when all of the following are true:

- You want to build a primary index and data subfile
- Your data is sorted on primary key
- The MIDAS file being built contains NO entries whatsoever: see Note below.

Use SECBLD: Use SECBLD when all of the following are true:

- You are building one or more secondary index subfiles
- Your input data is sorted on a secondary key field
- The secondary index subfile(s) to be built contain NO entries (See Note below.)

The input data must contain a copy of the primary key associated with each particular secondary key entry you want added to the file. SECBLD locates the primary key entry in the index subfile so the secondary index entry can be added.

PRIBLD and SECBLD are much faster than BILD$R, so try to use them whenever possible unless one or more of the above conditions are not true.

Note

If you're attempting to re-build an existing file that previously contained entries, make sure that each index you want to build from sorted data is truly empty, and does not contain obsolete pointers to data subfile entries that no longer exist. Use MPACK or KIDDEL to clean out the index subfiles before attempting to re-build them with sorted input data.

Use BILD$R: Use BILD$R when these conditions occur in the combinations indicated:

- Your input data is not sorted by the index to be built

and/or

- Your output (MIDAS) file already contains entries in the subfile to be built

Calls to PRIBLD, SECBLD and BILD$R can be made from the same program to build a single MIDAS file as long as you don't attempt to build the same index subfile from both BILD$R and PRIBLD or SECBLD at the same time.

Cautions

There are several important points you should keep in mind when using the offline file-building routines:

- Programs that make calls to BILD$R should beware of making calls concurrently to PRIBLD and/or SECBLD (and vice versa) because conflicts may occur when two routines try to access or modify the same index subfile. When this happens, the second routine to access that subfile will report an error and will abort unless an alternate return is provided by the program. (See the list of error messages that follows each routine.)

- Programs that use any of these routines to build a MIDAS file should not be run at the same time as application programs which access the same MIDAS file.

- A MIDAS file opened for use by any of the file-building routines cannot be in use by any other user or process, for reading or writing.

- PRIBLD, SECBLD, and BILD$R can only be used to build a single MIDAS (output) file. There is no capability for processing more than one output file through any of these file-building routines.

The remainder of this section describes the calling sequences and the functions of PRIBLD, BILD$R and SECBLD.

Event Sequence Flag

PRIBLD, SECBLD and BILD$R each use the same "flag" argument in their calling sequence. This flag, called segflag in the argument list, is used as a communications tool between the routine and the user. With it, you tell the routine when to start and stop processing. In return, the routine tells you the state of the build operation. It can have one of four values as shown in Table 14-4. When first calling one of these routines from a program to add an entry, supply a flag value of Ø in the calling sequence; this is essentially an initialization request to the routine. It tells the routine to start processing the data provided by your program. When the first record has been successfully processed, the routine sets the flag value to 1; the flag remains set at 1 until the last entry is processed. At this point, the user's program should issue a "finalization" request, setting the event sequence flag to 2. This is done by making another call to the routine in which seqflg has a value of 2; every other argument in the calling sequence, except for the unit and altrtn arguments, is ignored, and may have a value of Ø. A finalization request must be used to close the currently opened index subfile before another index subfile can be opened. When the finalization request is fulfilled, the routine will set the seqflg value to 3, indicating that the particular subfile has been closed.

Table 14-4. Event Sequence Flag Values

Value	Meaning
Ø	Set by user and passed to routine to signal that first record of input data is to be processed; this is essentially an initialization request.
1	Set by routine after first record has been processed. Remains set at 1 until user sets it to 2.
2	Finalization request: set by the user and passed to routine after last entry in input data has been processed.
3	Set by the routine to indicate that finalization is complete; if more than one index subfile is open, a separate finalization request must be made for each one. The routine acknowledges the closing of each index by setting the flag to 3.

Since these routines are all serially reusable, the build process can be restarted for a new index subfile by setting seqflg to Ø again (especially useful in BILD$R and SECBLD).

General Use of Sequence Flag: This is a very generalized example of how the process control flag is used:

```
C     INITIALIZATION REQUEST
      SEQFLG = Ø
      .
      .
      .
C     SET UP ARGUMENTS FOR CALL
      .
      .
      .
      CALL routine-name (SEQFLG, arguments.....)
C     FINISH UP
C     MAKE CLOSE REQUEST
      SEQFLG =2                              /*REQUEST TO CLOSE
      CALL routine-name (SEQFLG, arguments.....)
```

Error Handling

Although the alternate return feature is an acceptable method of handling errors, it is not consistent with the ever growing trend towards modular programming. The use of non-local "GOTO's" is frowned upon by all proponents of modular structure, and the use of flags to pass error status codes between the main program and the routines it calls is highly favored. Errors are then handled through a normal return to the calling program at which point the flag is checked and some sort of action is taken, usually by an on-unit (PL/I) or other exception handler.

Recording Errors: Errors that occur during the file-building process can be written to a disk file instead of appearing at the terminal. Use ERROPN to open and name such a file. Similarly, milestones can be recorded in this file by calling KX$TIM.

PRIBLD

The PRIBLD routine builds a primary index subfile and adds the corresponding data records to the data subfile. The input file must be sorted by primary key field and the MIDAS (output) file must be empty. This is because PRIBLD cannot add sorted primary key entries to an index subfile that already contains key values. PRIBLD is the fastest method of building a primary index subfile; if you're concerned about speed and performance, use PRIBLD instead of BILD$R whenever possible.

PRIBLD Calling Sequence

PRIBLD's calling sequence is:

CALL PRIBLD (seqflg, primkey, data, dlength, unit, altrtn, danum)

The arguments are all integer (INT*2) except for danum, which is a floating-point (REAL*4) number used in building direct access MIDAS files only.

seqflg The event sequence flag, see Table 14-4, above.

primkey A numeric variable or a one-dimensional array, which can be an integer or real number, depending on the key type, which contains the primary key value to use on this call.

data A one-dimensional array containing the data to be added. If dlength is zero, data may also be zero.

dlength The length of data in words. For fixed-length records, if dlength is less than the record size originally defined for the file, the entry written to the MIDAS file will be padded with Ø's. Excess data is ignored. For variable-length records, specify the exact length of the record being added.

unit The file unit on which the MIDAS file is opened.

altrtn The number of a statement in the program to be used as an alternate return; if Ø is supplied for the altrtn argument, control returns to PRIMOS in the event of an error. See Error Handlers, below.

danum The entry number for direct access files; this is a REAL*4 number. If the indicated entry slot is already occupied, the entry is added to the end of the data subfile. Specify a Ø for this argument if the MIDAS file being built is a keyed-index file.

PRIBLD Error Messages

When using PRIBLD, one has to be careful of other users accessing the file that is being built. If this happens, PRIBLD may not be able to recover from the error and the file will probably be garbled. If one of the following error messages is encountered, the user can call PRIBLD to finish building the index, (set SEQFLG to 2) and the file will be reasonably complete, except for the problem as indicated by the error message. If a file system error occurs (i.e., one that doesn't appear in the list below), you can be sure that the file will be damaged. In this case, zero the file with KIDDEL, attempt to figure out what happened, and try again. In both kinds of errors, PRIBLD usually prints out the key supplied by the user (i.e., the entry it was trying to process), along with the error message.

▶ CAN'T USE PRIBLD AND BILD$R SIMULTANEOUSLY

This is a fairly obvious message warning the user against simultaneous access to the primary index subfile. This can happen when the user has added one or more entries to the primary index with BILD$R and has now called PRIBLD which generates this message. Either continue adding entries or finish building index Ø with the appropriate calls to BILD$R, but not PRIBLD.

▶ ILLEGAL SEQFLG

The value of SEQFLG is incorrect. Remember, the first call to PRIBLD to add an entry must have a SEQFLG of Ø which PRIBLD will return as a 1. Subsequent calls to PRIBLD to add additional entries must continue to have a SEQFLG of 1. The final call to PRIBLD to close building index Ø for that MIDAS file must have a SEQFLG of 2 and PRIBLD will return it as a 3.

▶ NOT A VALID MIDAS FILE

The first time PRIBLD is called to add an entry (SEQFLG = Ø) to the primary index of an MIDAS file, PRIBLD calls KX$RFC both to verify that the file is indeed a valid MIDAS file and to gather certain configuration data needed to build the file. See Appendix A for a list of other messages that may be returned by KX$RFC through PRIBLD.

▶ INDEX Ø NOT ZEROED

The file must not contain any entries if PRIBLD is to be used to build the primary index. Use KIDDEL to zero the file.

▶ INDEX Ø: INDEX BLOCK SIZE GREATER THAN MAXIMUM DEFAULT

This is a fatal error that may occur on the first call to PRIBLD to add an entry. The argument RECLNT in the MIDAS paramter file KPARAM is less than the block size specified for some index level. Either fix the file or fix RECLNT in KPARAM and rebuild MIDAS before continuing. (See Section 15 for additional details.)

▶ KEY SEQUENCE ERROR

The key provided in the current call is less than or equal to the key provided in the previous call to PRIBLD.

▶ INDEX Ø: ±Ø.nnnnnnn E±nn INVALID DIRECT ACCESS ENTRY NUMBER

This error occurs during direct access file processing only. It can happen for one of three reasons:

1. The record number supplied was less than zero
2. The record number supplied was not a whole number
3. The supplied number exceeds the number of entries pre-allocated by CREATK

It is possible the user may have changed this number with CREATK and forgot to MPACK the file to effect the change. Try MPACKing the file.

▶ DATA SUBFILE FULL

No more entries may be added to the data subfile and therefore to the primary index. However, a call to PRIBLD to finish building the primary index (with SEQFLG = 2) may still be made.

SECBLD

The SECBLD routine builds secondary index subfiles from input data sorted by secondary key. The index subfile being built must not contain any entries prior to the calling of SECBLD. A copy of the primary key must be included as one of the arguments in the call so that SECBLD can make the appropriate connections between the data subfile entries already in the file and the secondary index entries being added.

When making calls to SECBLD in a program, avoid making calls to BILD$R that attempt to open the same secondary index subfile. If BILD$R already has the secondary index subfile open when SECBLD is called, SECBLD will return the error message:

▶ CAN'T USE SECBLD AND BILD$R SIMULTANEOUSLY

SECBLD Calling Sequence

The calling sequence of SECBLD is:

```
CALL SECBLD (seqflg,seckey,pkey,index,secdat,sdsiz,unit,altrtn)
```

There are no special arguments for direct access files; this is because the data entries have already been added and the record numbers need not be supplied in order to add secondary index entries. The complete argument list is given below. Data types are given in parentheses.

seqflg — The event sequence flag, described in Table 14-4 (INT*2).

seckey — The secondary key value to be added to the index subfile (INT*2).

pkey — The primary key value that references the same record as seckey (INT*2).

index — The secondary index subfile number being built during this call to SECBLD (INT*2).

secdat — The secondary data to be stored in this index entry (INT*2). This applies only to indexes for which the secondary data feature was chosen during index definition. Specify Ø if you don't want to add any secondary data for this particular index.

sdsiz — The size of the secondary data supplied in this call; specify in words (INT*2). Specify a Ø if you supplied Ø for the previous argument.

unit — The file unit on which the MIDAS file is open (INT*2).

altrtn — The number of the statement in the calling program to which control returns in the event of an error (INT*2). If Ø (no alternate return in program), SECBLD exits to PRIMOS when an error occurs.

SECBLD Error Messages

Like PRIBLD, SECBLD is fast and efficient but is not able to protect the file it is working on from damage caused by outside interference during the file-building process. Many of the error messages returned by SECBLD are quite similar to those returned by PRIBLD. The same error-handling method described under PRIBLD applies to SECBLD, so it will not be repeated here. The symbols "##" as used in the message representations below are replaced by a secondary index number when actually returned by SECBLD.

▶ INDEX ##: DOES NOT EXIST

The indicated index is either an invalid index number or simply doesn't exist in the MIDAS file. Either go back and ADD the index with CREATK and try again or remove all references to this index from the program.

▶ INDEX ##: CAN'T USE SECBLD AND BILD$R SIMULTANEOUSLY

This means that the user may have added one or more entries to this index with BILD$R and has now called SECBLD to add an entry to it. The user may continue adding entries or finish building this index with the appropriate calls to BILD$R, but not to SECBLD.

▶ INDEX ##: ILLEGAL SEQFLG

The value of SEQFLG is incorrect. See the discussion for this error under PRIBLD Error Messages, above.

▶ INDEX ##: NOT A VALID MIDAS FILE

This message is returned for the same reasons given for PRIBLD, above.

▶ INDEX ##: NOT ZEROED

The index must not contain any entries if you're trying to use SECBLD to build it. Use KIDDEL to zero this index or to zero the entire file.

▶ INDEX ##: INDEX BLOCK SIZE GREATER THAN MAXIMUM DEFAULT

See the discussion under PRIBLD Error Messages, for an explanation of this error message.

▶ INDEX ##: KEY SEQUENCE ERROR

The supplied secondary key is either less than the secondary key supplied in the last call to SECBLD for this index, or is a duplicate of the secondary key supplied in the last call to SECBLD for this index, and the index does not allow duplicates.

▶ INDEX ##: CAN'T FIND PRIMARY KEY IN FILE

SECBLD was unable to find the key value supplied for the pkey argument. The index subfile does not contain this value. When this error occurs, both the primary key (index Ø) and the secondary key supplied in the call to SECBLD are displayed with the above error message.

▶ INDEX ##: INDEX FULL

No more entries may be added to this particular secondary index, but a call to SECBLD to finish building this index (set SEQFLG to 2) may still be made.

BILD$R

BILD$R can be used to build the primary index subfile and data subfile, as well as any or all of the secondary index subfiles associated with a particular MIDAS file. BILD$R processes both sorted and unsorted data, and can add entries to files that already contain index entries in primary and/or secondary subfiles; it can also work on MIDAS files that are essentially empty.

Calls to BILD$R should not be made concurrently with calls to PRIBLD or SECBLD or the calling program will abort (in the absence of an alternate return).

BILD$R Calling Sequence

The calling sequence of BILD$R is:

```
CALL BILD$R (seqflg,key,pbuf,bufsiz,danum,index,unit,altrtn)
```

The arguments for BILD$R are described below. Data types are shown in parentheses.

seqflg	The event sequence flag; see Table 14-4 above (INT*2).
key	The primary or secondary key value to be added on this call (INT*2).
pbuf	If a primary index entry is being added, this is the data subfile entry only (INT*2). If adding a secondary index entry, this is the primary key that references the same data record as the secondary key entry being added. (If you're using secondary data, it should be placed in pbuf, immediately following the primary key value.)
bufsiz	The size of pbuf in words (INT*2). See Note below.
danum	The record entry number for direct access files; this is a REAL*4 number. If the indicated entry slot is already occupied, the entry is added to the end of the data subfile. Specify a Ø for this argument if the MIDAS file being built is a keyed-index file, or if adding a secondary index entry.
index	The number of the index subfile being built on this call to BILD$R (INT*2). (Supply Ø if building the primary index.)
unit	The file unit number on which the MIDAS file is opened (INT*2).

altrtn — The alternate return in the calling program to which control is passed in the event of an error (INT*2). If specified as Ø, the program will abort and return you to PRIMOS command level.

Note

When adding a primary index entry, the bufsiz argument is ignored unless the file contains variable length records. In this case, bufsiz represents the length of the data record only. When adding a secondary index entry, supply a Ø if you've already put the desired secondary data into pbuf. If non-zero, bufsiz indicates the total size of the primary key and the size of the secondary data supplied in pbuf. Extra secondary data is ignored and insufficient data is padded with Ø's. Note that this differs from the way PRIBLD and SECBLD treat a data size specification of Ø.

BILD$R Error Messages

Like PRIBLD and SECBLD, BILD$R cannot recover from simultaneous access errors, so the user should beware of them. Both file system errors and BILD$R-specific errors may be returned while using BILD$R to build a file. The BILD$R-specific ones are listed below. Where the error messages are the same as those given previously for PRIBLD and SECBLD, the reader is referred to those error message discussions.

For both types of errors, BILD$R will close all the internal files it has opened and usually will print the key supplied by the user along with the appropriate error message.

▶ INDEX ##: DOES NOT EXIST

The indicated index is either an invalid index number or doesn't exist in the MIDAS file. The user may go back and ADD the index with CREATK and try again or remove all references to this index from the program.

▶ INDEX ##: CAN'T USE BILD$R AND PRIBLD/SECBLD SIMULTANEOUSLY

This can happen when the user has added one or more entries to this index with PRIBLD or SECBLD and then calls BILD$R to add an entry to the same index. You can either continue adding entries or finish building this index by making the appropriate calls to either PRIBLD or SECBLD, but not to BILD$R.

▶ INDEX ##: ILLEGAL SEQFLG

The value of SEQFLG is incorrect; refer to the discussion of this under PRIBLD Error Messages above.

▶ INDEX ##: NOT A VALID MIDAS FILE

See the discussion for this under PRIBLD Error Messages.

▶ INDEX ##: INDEX BLOCK SIZE GREATER THAN MAXIMUM DEFAULT

See the discussion for this under PRIBLD Error Messages.

▶ INDEX Ø: DIRECT ACCESS FILE - INDEX OF -1 AND ENTRY # REQUIRED FOR PRIMARY KEY

The user is attempting to build the primary index of a direct access file. In such cases, an index number of -1, not Ø, must be used and a REAL*4 entry number must be supplied in array(3-4).

▶ INDEX Ø: ±Ø.nnnnnnn E±nn INVALID DIRECT ACCESS ENTRY NUMBER

See explanation under PRIBLD Error Messages above.

▶ INDEX ##: KEY SEQUENCE ERROR

The supplied key is a duplicate of a key already entered in the indicated index and the index does not allow duplicate entries. BILD$R does check for the case where there is an entry already present in an index that does not allow duplicates to see if this entry points to a deleted record. In such cases, the new entry will be made on top of the old entry.

▶ INDEX ##: CAN'T FIND PRIMARY KEY IN FILE

See SECBLD Error Messages above.

▶ INDEX ##: INDEX FULL

No more entries may be added to this particular index, but a call to BILD$R to close building this index (seqflg = 2) may still be made.

▶ INDEX Ø: DATA SUBFILE FULL

No more entries may be added to the primary index, but a call to BILD$R to finish building the primary index (seqflg = 2) may still be made.

Offline Routine Example

Suppose an airline has a file containing flight number, origin, destination, departure time, arrival time, etc. information. Each record has several fields containing this information, but there are no keys by which the file can be searched. Furthermore, the airline desires to search on a key that is a combination of several fields, for example, flight number, origin and destination. The airline decides to put this information into a MIDAS file. The flight number, origin and destination fields will be concatenated to form the primary key, and the date, departure time, and arrival time fields will become secondary keys in the MIDAS file. Because KBUILD cannot handle concatenated keys, the offline file-building routines are the ideal tools for building the MIDAS file. The original sequential disk file is used for input, and it is listed below, along with a COMOUT file of the CREATK session in which the MIDAS template used in this application was set up. The example program uses all three file-building routines, as some fields in the input file are sorted and others aren't.

The Input File: The sequential input file, called INPUT_FILE, is made up of eight fields:

```
195 BOS LOG NWK EWR 11/Ø5/8Ø 12:ØØ 13:ØØ
2Ø5 BOS LOG NYC JFK 11/Ø5/8Ø 12:15 12:5Ø
3Ø5 BOS LOG NYC LGA 11/Ø5/8Ø 12:3Ø 13:1Ø
696 CHI ORD WOR WOR 11/Ø5/8Ø 1Ø:45 12:15
1Ø6 NWK EWR ORL ORL 11/Ø6/8Ø Ø8:4Ø 11:55
749 NYC LGA CHI ORD 11/Ø5/8Ø 16:ØØ 18:3Ø
65Ø WOR WOR BOS LOG 11/Ø5/8Ø 12:45 13:15
```

The only fields that are in sorted order are the origin and date fields.

The MIDAS Template: The MIDAS file, called FLIGHTS, has four keys, which are defined in the following CREATK session:

```
OK, CREATK
[CREATK rev 17.6]

MINIMUM OPTIONS? YES

FILE NAME? FLIGHTS
NEW FILE? YES
DIRECT ACCESS? NO

DATA SUBFILE QUESTIONS

PRIMARY KEY TYPE: A
PRIMARY KEY SIZE = : B 9
DATA SIZE = : 20

SECONDARY INDEX

INDEX NO.? 1

DUPLICATE KEYS PERMITTED? YES

KEY TYPE: A
KEY SIZE = : B 8
SECONDARY DATA SIZE = : 0

INDEX NO.? 2

DUPLICATE KEYS PERMITTED? YES

KEY TYPE: A
KEY SIZE = : B 5
SECONDARY DATA SIZE = : 0

INDEX NO.? 3

DUPLICATE KEYS PERMITTED? YES

KEY TYPE: A
KEY SIZE = : B 5
SECONDARY DATA SIZE = : 0

INDEX NO.? (CR)

SETTING FILE LOCK TO N READERS AND N WRITERS
```

The AIRLINE Program: The program that builds the FLIGHTS file from the data in INPUT_FILE is AIRLINE:

```
C     AIRLINE PROGRAM
C
C     THIS PROGRAM BUILDS A MIDAS FILE FROM A SEQUENTIAL DISK FILE
C     USING THE OFFLINE ROUTINES PRIBLD, SECBLD AND BILD$R.
C     THE PRIMARY KEY IS MADE UP OF THREE FIELDS FROM
C     THE INPUT FILE AND IS THUS A CONCATENATED KEY.
C
      INTEGER*2 FILNAM(40),                 /* FILE NAME BUFFER
     +          IFUNIT,                     /* INPUT FILE FUNIT
     +          MFUNIT,                     /* MIDAS OUTPUT FILE FUNIT
     +          PSQFLG,                     /* PRIBLD'S SEQFLG
     +          SSQFLG,                     /* SECBLD'S SEQFLG
     +          BSQFLG(2),                  /* BILD$R'S SEQFLG'S
     +          INPREC(20),                 /* INPUT RECORD BUFFER
     +          PRIKEY(5),                  /* PRIMARY KEY BUFFER
     +          SECKEY(3),                  /* SECONDARY KEY BUFFER
     +          CHRPOS(2),                  /* POSITION   SIZE ARRAY FOR TSRC$$
     +          ERRCOD,                     /* ERROR CODE
     +          I
C
$INSERT SYSCOM>KEYS.F
C
$INSERT SYSCOM>ERRD.F
C
C
C---INPUT AND OPEN THE INPUT FILE
C
5     CALL TNOUA ('ENTER INPUT FILE NAME: ', 23)
      READ (1, 10) FILNAM
10    FORMAT (40A2)
      CHRPOS(1) = 0
      CHRPOS(2) = 80
      IFUNIT = 0
      CALL TSRC$$ (K$READ + K$GETU,
     +             FILNAM, IFUNIT, CHRPOS, I, ERRCOD)
      IF (ERRCOD .EQ. E$FNTF) GOTO 5
      IF (ERRCOD .NE. 0) GOTO 9000
      CALL ATTDEV (IFUNIT, 7, IFUNIT, 80)   /* TELL IOCS ABOUT DISK FILE   UNIT
C
C---INPUT AND OPEN THE MIDAS OUTPUT FILE.
C
C   NOTICE THAT (1) IT IS OPENED FOR READING AND WRITING (K$RDWR), AND
C               (2) WE DO NOT (!) CALL NTFYM$ OR OPENM$/CLOSM$
C
20    CALL TNOUA ('ENTER MIDAS OUTPUT FILE NAME: ', 30)
      READ (1, 10) FILNAM
      CHRPOS(1) = 0
      CHRPOS(2) = 80
      MFUNIT = 0
      CALL TSRC$$ (K$RDWR + K$GETU,
     +             FILNAM, MFUNIT, CHRPOS, I, ERRCOD)
      IF (ERRCOD .EQ. E$FNTF) GOTO 20
      IF (ERRCOD .NE. 0) GOTO 9000
C
```

```
C---INIT SEQFLG'S
C
      PSQFLG = Ø                              /* PRIBLD SEQFLG FOR INDEX Ø
      SSQFLG = Ø                              /* SECBLD SEQFLG FOR INDEX 1
      BSQFLG(1) = Ø                           /* BILD$R SEQFLG FOR INDEX 2
      BSQFLG(2) = Ø                           /* BILD$R SEQFLG FOR INDEX 3
C
C---MAIN LOOP TO READ A RECORD FROM THE INPUT FILE AND MAKE
C   THE APPROPRIATE CALLS TO PRIBLD, SECBLD, AND BILD$R TO ADD THE DATA
C   RECORD AND VARIOUS ENTRIES.
C
1ØØ   READ (IFUNIT, 11Ø, END = 5ØØ) INPREC  /* READ THE INPUT RECORD
11Ø   FORMAT (2ØA2)
C
C.....BUILD THE PRIMARY KEY -
C        A CONCATENATION OF THE ORIGIN, DESTINATION,   FLIGHT NUMBER.
C
      PRIKEY(1) = INPREC(3)
      PRIKEY(2) = LT (INPREC(4), 8) + RS (INPREC(7), 8)
      PRIKEY(3) = LS (INPREC(7), 8) + RS (INPREC(8), 8)
      PRIKEY(4) = INPREC(1)
      PRIKEY(5) = LT (INPREC(2), 8)
C
      CALL PRIBLD (PSQFLG, PRIKEY,          /* ADD THE PRIMARY KEY + DATA RECORD
     +             INPREC, 2Ø, MFUNIT, Ø, Ø)
C
C......ADD SECONDARY KEY 1 - THE DATE.
C         SINCE IT IS SORTED, WE USE SECBLD.
C         ALSO, SINCE IT IS WORD ALIGNED, WE DON'T HAVE TO MOVE THE
C         KEY TO THE BUFFER 'SECKEY'.
C
      CALL SECBLD (SSQFLG, INPREC(11),
     +             PRIKEY, 1, Ø, Ø, MFUNIT, Ø)
C
C.....ADD SECONDARY KEY 2 - THE DEPARTURE TIME.
C        IT IS UNSORTED, SO WE CALL BILD$R AND IS UNALIGNED, SO
C        WE MOVE IT TEMPORARILY TO THE BUFFER 'SECKEY'.
C
      SECKEY(1) = LS (INPREC(15), 8) + RS (INPREC(16), 8)
      SECKEY(2) = LS (INPREC(16), 8) + RS (INPREC(17), 8)
      SECKEY(3) = LS (INPREC(17), 8)
C
      CALL BILD$R (BSQFLG(1),               /* NOTE BSQFLG(1) IS FOR INDEX 2.
     +             SECKEY, PRIKEY, Ø, Ø, 2, MFUNIT, Ø)
C
C.....ADD SECONDARY KEY 3 - THE ARRIVAL TIME.
C        THIS FOLLOWS THE SAME PATTERN OF MOVING THE KEY AND
C        CALLING BILD$R AS WE DID WITH SECONDARY KEY # 2.
C
      SECKEY(1) = LS (INPREC(18), 8) + RS (INPREC(19), 8)
      SECKEY(2) = LS (INPREC(19), 8) + RS (INPREC(2Ø), 8)
      SECKEY(3) = LS (INPREC(2Ø), 8)
C
      CALL BILD$R (BSQFLG(2),               /* NOTE BSQFLG(2) IS FOR INDEX 3)
```

```
     +            SECKEY, PRIKEY, Ø, Ø, 3, MFUNIT, Ø)
C
      GOTO 1ØØ                              /* LOOP ON READING   ADDING ENTRIES
C
C---INPUT FILE IS EXHAUSTED.
C        SET THE SEQUENCE FLAG TO '2' AND MAKE A FINAL CALL TO THE
C        APPROPRIATE ROUTINE FOR EACH INDEX BEING BUILT; THEN CLOSE
C        THE INPUT AND OUTPUT FILES.
C
5ØØ   PSQFLG = 2
      CALL PRIBLD (PSQFLG, Ø, Ø, Ø, MFUNIT, Ø, Ø) /* FINALIZE PRIMARY KEY
C
      SSQFLG = 2
      CALL SECBLD (SSQFLG, Ø, Ø, 1, Ø, Ø, MFUNIT, Ø) /* FINALIZE SECONDARY KEY 1
C
      BSQFLG(1) = 2
      CALL BILD$R (BSQFLG(1), Ø, Ø, Ø, Ø, 2, MFUNIT, Ø) /* FINALIZE SEC. KEY 2
C
      BSQFLG(2) = 2
      CALL BILD$R (BSQFLG(2), Ø, Ø, Ø, Ø, 3, MFUNIT, Ø) /* FINALIZE SEC. KEY 3
C
      CALL SRCH$$ (K$CLOS, Ø, Ø, IFUNIT, I, ERRCOD) /* CLOSE INPUT FILE
      IF (ERRCOD .NE. Ø) GOTO 9ØØØ
C
      CALL SRCH$$ (K$CLOS, Ø, Ø, MFUNIT, I, ERRCOD) /* CLOSE MIDAS OUTPUT FILE
      IF (ERRCOD .NE. Ø) GOTO 9ØØØ
C
      CALL EXIT                             /* EXIT TO PRIMOS
C
C---ERROR HANDLER
C     TAKES THE BRUTE FORCE APPROACH OF CLOSING INPUT   OUTPUT FILES,
C     IGNORING ANY ERRORS ENCOUNTERED,   EXITING WITH A CALL TO ERRPR$.
C
9ØØØ  CALL SRCH$$ (K$CLOS, Ø, Ø, IFUNIT, I, I) /* CLOSE INPUT FILE
C
      CALL SRCH$$ (K$CLOS, Ø, Ø, MFUNIT, I, I) /* CLOSE MIDAS OUTPUT FILE
C
      CALL ERRPR$ (K$NRTN, 'EXAMPLE', 6, Ø, Ø)
C
      END
```

Sample Output: When the program is run, the user simply enters the names of the input and output (MIDAS) files and the offline routines take care of the rest. For example:

```
OK, SEG #AIRLINE
ENTER INPUT FILE NAME: INPUT FILE
ENTER MIDAS OUTPUT FILE NAME: FLIGHTS

PRIMARY INDEX AND DATA

SECONDARY INDEX     1
Index Ø: Entries indexed:         7
Index  1: Entries indexed:         7
Index  2: Entries indexed:         7
Index  3: Entries indexed:         7
OK,
```

PART III. OTHER MIDAS ROUTINES

OTHER OFFLINE ROUTINES

Several internal routines previously available only to R-mode users in earlier versions of MIDAS are now generally available to all users. They are limited in usefulness, and may appeal only to FORTRAN users who are also making use of the offline routines discussed earlier in this section. The routines covered here are: ERROPN, FILERR, FILHER, and KX$TIM. Of these four routines, two are still used internally by MIDAS: ERROPN and KX$TIM. FILHER and FILERR, no longer used by MIDAS, have been kept for compatibility. All of these routines are described briefly in Table 14-5.

Table 14-5. Other Offline Routines

Routine	Function	Comments
ERROPN	Opens a logging file to record errors and milestone statistics (see KX$TIM, below)	Used by KBUILD to open and name error/logfile
FILERR	Sets up and writes error messages to terminal	No longer used by MIDAS, but still available to users
FILHER	Converts MIDAS error code to corresponding text and passes it to FILERR	No longer used by MIDAS, but still available to users
KX$TIM	Prints milestone statistics, including CPU, disk and wall clock time elapsed since last milestone -- statistics are written to err/log file if ERROPN was called to open it	Used by KBUILD to generate milestones

ERROPN

ERROPN is a routine used to open an error/logging file. Its calling sequence is:

```
CALL ERROPN (funit)
```

funit is the INT*2 file unit on which the error/logging file is to be opened. It is returned as Ø if no file was opened. MUST BE A VARIABLE.

What It Does: ERROPN asks the user for a pathname for the error/logging file with the question:

```
ENTER LOG/ERROR FILE NAME:
```

If you enter just a carriage return or blank line, funit is set to Ø and ERROPN returns. If a valid pathname is entered, a new SAM file is created if necessary and opened for writing on the PRIMOS file unit returned in funit (via the key K$GETU), and truncated. If an error occurs on the "open" call, the user is asked to enter another pathname. The purpose of the truncate operation is to make sure that the file is empty, in case the indicated file already exists. If an error occurs on the truncate operation, it is noted with the message: "COULDN'T TRUNCATE LOG/ERROR FILE" and ERROPN returns.

How ERROPN Works: The file unit number (funit) on which the error/logging file is opened is stored in an internal common area called /ERRFIL/. If any of the offline routines generates an error message, or if KX$TIM is called to print a milestone, the error message or the milestone is sent to the error/log file opened on /ERRFIL/ as well as to the terminal.

/ERRFIL/ only remembers the last error/logging file opened and does not notice that the user may have in fact closed the file in the meantime. Errors arising from attempts to write to this file are taken as a sign that the file has been closed and are therefore ignored.

On page 14-34 under ERROPN: the *iunit* argument should always be specified as a variable and not a constant, because it returns a value of 0 if the call to ERROPN is unsuccessful.

FILERR

FILERR is a user-callable routine that prints out the text of an error message which corresponds to a code returned by one of the system routines. Its calling sequence is:

```
CALL FILERR (caller-name, optmsg, msglen, altrtn)
```

The arguments shown above, their meanings and their data types are:

caller-name	Six-character name of calling routine (INT*2)
optmsg	Optional error message to print out (INT*2)
msglen	Length of optmsg in characters (INT*2)
altrtn	Statement number of alternate return or Ø (no alternate return) (INT*2)

How It Works: FILERR indirectly calls the system routine ERRPR$ to print out the system error message corresponding to an error code returned by one of the system routines. The optional message, optmsg, is a user-supplied message to be printed along with the system-supplied one. The error code, understood to be in the variable CODE in the common area /CODE/ (defined in KPARAM), is translated into an actual error message by the internal MIDAS routine ERRTD$. The resulting error message, along with an optional error message (if its length is greater than Ø), plus the caller's name, is printed out at the terminal and to the error/logging file if there is one in the same format as would ERRPR$. If altrtn is Ø, FILERR exits to PRIMOS, otherwise it returns normally to the caller.

Note

This routine is of limited use considering that users can get at the common area /CODE/ only by using the unshared V-mode MIDAS library NVKDALB.

FILHER

FILHER is a user-callable routine to convert and print a MIDAS error code to ASCII form, calling FILERR to print it out. If you want to use PRIBLD, SECBLD or BILD$R, and you want to open your own error/logging files, all three of these internal routines should be used together. FILHER's calling sequence is:

```
CALL FILHER (errcod, altrtn)
```

errcod is an INT*2 argument representing a MIDAS error code, and altrtn is a user-supplied alternate return.

How It Works: If errcod is less than or equal to 13, or if there is a log/error file opened (see ERROPN), FILERR is called with /CODE/ set to E$NULL (null error message), the optional message is set to "FILE HANDLER ERROR xx", where xx is the ASCII representation of errcod, and the alternate return in the calling sequence to FILERR set to altrtn. If FILERR returns to FILHER, then FILHER returns normally to the user.

If errcod is greater than 13 and /ERRFIL/ indicates that an error/logging file is not open, FILHER will take an alternate return through altrtn or, if that is Ø, it will return through ERRPR$ with a null error message.

KX$TIM

KX$TIM is a user-callable routine that displays milestones for the offline file-building routines PRIBLD, SECBLD and BILD$R. These milestones are displayed at the terminal and optionally recorded in an error/logging file opened by ERROPN.

Its calling sequence is:

```
CALL KX$TIM (numrec, optmsg, msglen)
```

numrec Indicates the number of records processed for this milestone (INT*4). Special case values of Ø and -1 make it possible to generate headers and so forth -- details below.

optmsg Supplied by the user only if desired (INT*2). If supplied, the length of optmsg, in words, must be passed in msglen.

msglen The length of the optional optmsg, in characters (INT*2). Set it to Ø if there is no optional message.

How KX$TIM Works: First, if there is an optional message, it is printed to the terminal and to the optional error/logging file. Then a "milestone" consisting of numrec, date and time, number of CPU minutes used since the last call to KX$TIM, number of disk I/O minutes used since the last call to KX$TIM, total CPU and disk time used so far, and the difference in the total since the last call, is printed in a similar fashion. If numrec has a value of Ø, all counters will be initialized to Ø and a header will be printed out before the milestone line. By using a numrec value of -1, a milestone of Ø without a header or initialization, can be generated.

For a description of the milestones generated by KX$TIM, see Section 3. KBUILD itself calls KX$TIM to produce milestones during file-building.

SECTION 15

ADVANCED USES OF MIDAS

INTRODUCTION

This section is intended for users who are interested in modifying MIDAS parameters in hopes of obtaining better performance or in handling problems that are beyond the scope of standard MIDAS defaults. Users who want to use the extended options feature of CREATK or to explore the concept of index levels will also find this section helpful. It also describes what is in the MIDAS parameter file, KPARAM, and which parameters can be modified by the user.

Section Topics

Topics covered in this section are presented in this order:

- CREATK's extended options path -- dialog and description
- Double-length indexes -- using the version of CREATK that was created by C_LCREATK
- Adding new secondary indexes -- CREATK's ADD option
- Altering data subfile record length -- CREATK's DATA option
- Extending segment subfile length -- CREATK's EXTEND option
- Modifying template parameters -- CREATK's MODIFY option
- Modifying MIDAS defaults -- KPARAM's user-modifiable parameters

CREATK'S EXTENDED OPTIONS PATH

Most users will find the minimum options path of CREATK quite suitable for their needs and may want to limit the amount of space allocated for each index subfile. CREATK's extended options path offers a way to alter, on a per-file basis, some of the default file parameters used by CREATK in initializing MIDAS files.

The Purpose of Extended Options

The extended options feature of CREATK allows the user to specify the size of an index block at each index level in the index subfile. Index levels are described in detail under Index Levels, below. Generally, an index block contains key entries that point to records in the data subfile. An index subfile block entry includes:

- A key value, supplied by the user during data entry (file building)
- A three-word pointer to a data subfile record (in keyed-index access files)
- A five-word pointer to a data subfile record (in direct access files)
- Secondary data in secondary indexes (optional)

Defining Block Size

Under minimum options, CREATK doesn't ask for a block size. Instead, it assumes the default value of 1Ø24 words per block. Space may be wasted with 1Ø24 words per block, depending on the average index entry length. For instance, short files with small keys may be able to get by with smaller index blocks. On the other hand, increasing block size may improve access time by reducing the height (number of levels) in an index tree. (The fewer number of levels to search, the faster the access time.) By modifying the block size, the user may be able to optimize access time and the use of storage space.

The default setting of 1Ø24 words per block may waste space even if long keys and secondary data are used. So it may be worthwhile to reduce block size in many cases to economize resource usage. However, bear in mind that search-efficiency is increased by keeping block size large enough to hold a maximum number of entries without wasting space. Such an ideal index would be "dense," with many entries packed into fewer levels than there would be if the block size was smaller. In addition, larger blocks probably mean fewer blocks. The fewer the blocks that MIDAS has to search through, the better the performance. It is faster to search through one block with many entries than to search through many shorter index blocks. Keep in mind that 1Ø24 words is the physical disk record size, and I/O is more efficient when physical and logical block sizes match. Also, if block sizes are made too small, frequency of block splitting can degrade performance.

Block Size Specifications

The block size at the first, second, and last index levels (see Index Block Levels, below) can be changed with the extended options version of CREATK. The minimum acceptable block size must be at least large enough to hold 6 or 1Ø control words and two entries at that particular

level. The maximum block size is 32767 words. The minimum required block size varies with level: the last level index block always has 10 control words, while upper levels have 6 control words. CREATK checks to see whether or not your proposed block size will accommodate the minimum number of entries and control words, and lets you know whether it's acceptable. Last level index blocks also contain entry numbers if the file is direct access and, in secondary index subfiles that support the secondary data feature, last level index blocks also contain secondary data. This should be taken into account when modifying the block size.

If desired, the block size can be changed globally by modifying the RECLNT parameter in the KPARAM file; see MODIFYING MIDAS below. Once changed, this size will be applied to every file subsequently created with CREATK. However, existing files will retain the block size specified for them when they were created. It is recommended that if the block size is changed, it should be specified as a power of 2. Changing this parameter, however, is not guaranteed to buy you anything.

Index Block Levels

All of the entries in an index subfile are contained in blocks, and each block is associated with an "index level." When space is first allocated for an index subfile, that subfile has only one index level, which is called the "last" level. As the file grows in size and complexity, more index levels are created to help maintain search efficiency. Blocks in these index levels are collectively called "upper level" index blocks.

Why Index Levels?: The purpose of multi-level indexing is to maximize search and access efficiency. The important thing to remember here is that only the last level index blocks contain the index entries that consist of key values plus pointers to record entries in the data subfile.

As mentioned before, a "new" index subfile has only one level of indexing, the last level. At first, it contains only one block. Entries in this block point directly to the appropriate data subfile records. When this index block becomes full, it is split in half, producing two index blocks that are each half-filled. Each of these blocks is the same size as the original block. Another index level is then created above the last level index blocks. This newly-created "upper" index level has one block to start with, and it initially contains two entries that point to the blocks in the last level. When MIDAS searches for a particular index entry in an index subfile, it starts with the topmost index level and, using a search algorithm, follows special pointers that tell MIDAS which block in the next level down contains the desired index entry. As the file grows larger, the index blocks at various levels will become full and must be split in a similar fashion. Such splitting may require the creation of additional index levels so that the search process will not be impaired. Multi-level indexing refines the search process by eliminating the need

for MIDAS to search through every block at each index level in order to find the object of the search, which is always located in a last level index block.

Extended Options Dialog

To enter the extended options path of CREATK, answer "no" to the "MINIMUM OPTIONS?" prompt at the beginning of the CREATK dialog. The annotated dialog follows.

	Prompt	Response
1.	MINIMUM OPTIONS?	Enter N[O].
2.	FILE NAME?	Enter name of file to be created or name of existing file to be modified or examined.
3.	NEW FILE?	Enter Y[ES]. (Enter NO to obtain information about an existing file template.) See below.
4.	DIRECT ACCESS?	Enter Y[ES] or N[O], depending on whether the direct access feature is desired or not.
	DATA SUBFILE QUESTIONS	
5.	PRIMARY KEY TYPE:	Enter A,B,I,L, or S; same as for minimum options. (See Table 2-2 in Section 2.)
6.	PRIMARY KEY SIZE=:	Enter Bnn or Wnn, where nn is the number of bits or bytes or words that the key should contain; same as for minimum options dialog.
7.	DATA SIZE=:	For fixed-length records, enter record length. Include the key size in this figure for COBOL files.

8.	NUMBER OF ENTRIES TO ALLOCATE?	Asked only if you answered YES to "DIRECT ACCESS?" prompt above. Enter number of records for which to reserve room in the data subfile.
9.	FIRST LEVEL INDEX BLOCK SIZE=:	Enter number of words per block desired (default is 1024). See Defining Block Size, above.
10.	SECOND LEVEL INDEX BLOCK SIZE=:	Should be same as above response.
11.	LAST LEVEL INDEX BLOCK SIZE=:	Should be same as above.
	SECONDARY INDEX	Indicates beginning of secondary index subfile questions.
12.	INDEX NO?	Enter number from 1-17, or simply hit (CR) if no secondary indexes are needed.
13.	DUPLICATE KEYS PERMITTED?	Enter Y[ES] or N[O].
14.	KEY TYPE:	Same as for minimum options. See Table 2-2 in Section 2.
15.	KEY SIZE=:	Returned only if A and B type key is specified above. Enter Bnn or Wnn as described above.
16.	SECONDARY DATA SIZE=:	Specify secondary data size (FTN only) or hit (CR).
17.	FIRST LEVEL INDEX BLOCK SIZE=:	Enter desired number of words per block. See above.
18.	SECOND LEVEL INDEX BLOCK SIZE=:	Same as above.
19.	LAST LEVEL INDEX BLOCK SIZE=:	Same as above.

2Ø. INDEX NO?	Enter index number from 2-17 or hit carriage return if no more secondary indexes are needed. Prompts 13-19 are repeated if an index number is supplied. If you simply hit (CR), the dialog ends, and this prompt appears:

SETTING FILE LOCK TO N READERS AND N WRITERS

If the double-length index feature has been enabled, the prompt:

DOUBLE LENGTH INDEX?

will appear immediately after the "SECONDARY DATA SIZE" prompt in the dialog above. Double-length indexes are described in the following paragraphs.

DOUBLE-LENGTH INDEXES

For very large files that require index subfiles longer than the default of 1Ø segment subfiles per index, MIDAS offers the "double-length index" feature. If double-length indexes are desired, CREATK must be modified to include the double-length index request. There is a command file in MIDAS>SOURCE called C_LCREATK that creates a special version of CREATK allowing double-length indexes to be specified on a per-file basis. Note that the default version of CREATK does not ask if the user wants double-length indexes but instead assumes all indexes are the default length. However, files created under a long-index version of CREATK are supported by the default version of CREATK.

A double-length index, also called a "long index", is made up of two "regular" index subfiles that are 1Ø segment subfiles in length. Thus, each double-length index really contains 2Ø segment subfiles.

Specification of double-length indexes is selective; that is, the user decides which indexes will be double-length and which will be the default length. This option requires a bit of caution on the users's part because each double-length index takes up two single-length indexes. For example, if you define the primary key as a double-length index, you cannot define a secondary index 1; the next available index would be secondary index 2. If secondary index 2 is defined as a double-length index, the next available index would be secondary index 4, not index 3. Bear in mind, however, that secondary index 17 cannot be defined as a double-length index or else it would overwrite the data subfile segments.

MODIFYING A TEMPLATE

CREATK offers four file-modification options for changing an existing file template. These options should be used only when necessary to increase index subfile length, to alter the data subfile length, or to add a new secondary index. Although it is possible to change block size, secondary data size, duplicate support, and so forth, it is not possible to change key length or key type without recreating the file or index from scratch.

These options are invoked like all the other CREATK "old file" options, as explained in Section 12.

Note

> Don't forget that any of the changes made to a MIDAS file with these options won't be put into effect until the MPACK utility is run on the file. MPACK is documented in Section 12.

Adding Secondary Indexes

The ADD function allows a new secondary index to be added to an existing file. The dialog is similar to that used in creating a secondary index during template creation. If the secondary index already exists, an error message is displayed. Remember, only 17 secondary indexes can be defined per file. Be careful if you've already defined double-length indexes for the file!

Changing Record Length

The DATA function alters the length of a data subfile entry. This is the same thing as saying it alters the data size or record size in a MIDAS file. The DATA dialog is similar to the data subfile questions asked during template creation. Remember, the new data size will not go into effect until you MPACK the file. (Use the DATA option of MPACK.) When MPACK is run on the modified file, existing records in the file will be suitably truncated or padded with Ø's to make them compatible with the new data size.

Extending the Subfile

The EXTEND function allows the user to change the length of the segment subfiles and the length of the segment directory itself. The user is asked to supply the segment directory length in [illegible] and the segment subfile (index) length in words. If a Ø or (CR) is supplied, CREATK default values are used. The minimum subfile size is 64K words. The minimum segment directory size is 185 (segments), but this reflects a file with only one segment allocated for the data subfile. As the file grows, it will use up more segments. The current segment number at which the data subfile ends is called the "data growth point."

see PTU89

NUMBER OF SEGMENT SUBFILES

Modifying Other Template Parameters

The MODIFY function enables a user to change the following parameters in an existing MIDAS file template:

- Index block length (only if you're using non-minimum options path of CREATK)
- Secondary data size - see Note below
- Support for duplicate key occurrences
- Changing a single-length to a long (double-length) index, and vice-versa (if supported by the CREATK version in use). This is only permitted if you're not already using up all the available index subfile slots. In fact, CREATK doesn't even ask if you want to make an index a double-length one if it would prove physically impossible.

If the index you're attempting to MODIFY doesn't exist, an error message will be displayed.

Note

When secondary data size is modified for a particular index, the existing secondary data entries will be truncated or padded with Ø's when the file is MPACKed. This ensures that all the secondary data entries in that index will conform to the new secondary data size.

MODIFYING MIDAS

All of the user-modifiable MIDAS parameters are stored in the file KPARAM, located in the UFD MIDAS. This file contains all the parameters used by the MIDAS routines, but not all of them are subject to user modification. Modifiable parameters appear at the beginning of

Page 15-8 discusses the EXTEND option of CREATK as a method of making the index subfile longer. This method is preferable to the double-length index method which can also be used to lengthen an index subfile. Users are urged to use the EXTEND option whenever they need to enlarge an index subfile. Please note this carefully, as it was not explicitly stated anywhere in the book.

On page 15-9, in the section on RECLNT, the third sentence should read: "Users may change this value depending on what type of disk they are using." The following paragraph should be inserted between the two existing paragraphs in the RECLNT section:

The RECLNT parameter must be large enough to ensure that each index block can fit two key/pointer entries plus the 10 control words required in the last block level. However, RECLNT should not be given a value greater than 4095.

the KPARAM file, after the variable and constant declarations. The line separating parameters that users can modify from parameters that are fixed (not user-modifiable) reads: THE REMAINING PARAMETERS ARE FIXED. Refrain from tampering with the parameters that appear below this line, as the consequences are likely to be unpleasant.

Effecting Changes

Any changes made to parameters in KPARAM require that MIDAS be completely rebuilt by running the appropriate command files (see Section 13). R-mode users and users of unshared libraries should reload their program runfiles with the new version of the MIDAS library in order to take advantage of the changes made in the KPARAM file. Once MIDAS is rebuilt with the "new" KPARAM file, any files subsequently created with CREATK will have the new KPARAM-specified default measurements.

RECLNT: Index Block Length

The RECLNT parameter specifies the default index block size (length) in words, for each MIDAS file. As delivered, the RECLNT parameter is set to 1Ø24 words. Users may change this value depending on what type of disk they are using. Most Prime users have storage module disks with default physical record sizes of 1Ø24 words. The physical record size is also called the physical index block size. Index block sizes should be specified in integral multiples or fractions of the physical disk record size for optimal performance.

see PTU89

Remember that extended options (CREATK) can be used to change the block size on a per-file basis. If there are only a few files for which the index block size needs adjustment, it may be easier to use extended options on each one than to change RECLNT and then rebuild all of MIDAS. If you want this change applied to existing MIDAS files, you can do a MODIFY on each index in every file, and then MPACK each file. Alternatively, you can re-create each file with the just rebuilt version of CREATK.

SEGLNT: Segment Directory Length

The length of a MIDAS segment directory is set to 512 segment subfiles. Every index subfile is allowed 1Ø segment subfiles, also called "segments," albeit incorrectly, and the data subfile can have up to 327 segment subfiles, assuming that SEGLNT is set to 512. See Figure B-1 for a representation of a MIDAS file as a segment directory. This parameter may be increased to allow more entries per data subfile. The segment directory length can also be changed on a per-file basis with the EXTEND option of CREATK. If you want any change made to this parameter to be applied to existing MIDAS files, re-create them with the newly-rebuilt version of CREATK to make them compatible with the newly-created ones.

IWRAP: Segment Subfile Size

IWRAP specifies the number of words per segment subfile. The default setting of this parameter is 524288 words. This segment subfile length enables a single file system DAM file index level to handle up to 524288 words (on a storage module disk only).

This value can be increased to allow more entries per index, or it can be decreased if you want to limit the number of entries that can fit in an index subfile. Do not set IWRAP to a value lower than 64K words. Increasing the value of IWRAP is a good idea if you don't (and can't) have double-length indexes and you need more room for entries in an index subfile. However, in most cases, changing this parameter will not alter performance significantly, so leave it alone unless you know what you are doing. To make this change apply to files existing prior to the modification of IWRAP, re-create them with the newly-rebuilt version of CREATK.

BREAKI: Program Interrupt Control

BREAKI is set to a value of 1, indicating that MIDAS has control over when keyboard interrupts are allowed during MIDAS file processing. In the default break-handling method, user-initiated breaks, caused by hitting CTRL-P or BREAK, are disabled during calls to on-line MIDAS routines. MIDAS then re-enables breaks after these operations are completed. Users can control the enabling and disabling of breaks themselves by making calls to BREAK$. If BREAKs were disabled already when MIDAS was called, MIDAS will leave them disabled when control returns to the user. If BREAKI is set to Ø, MIDAS will not disable breaks, which can be a problem, especially if a BREAK occurs during file update.

RECYLA: Recycle Control

The RECYLA parameter is of interest only to people who are not using the current method of concurrent process handling as explained in Section 13. Possible reasons for this might be because they haven't upgraded their programs intentionally or because they are operating over networks, in which case the concurrent process handling method of Rev 17.6 will not work. In some cases, users may not have upgraded to the current version of MIDAS. In any of these events, MIDAS defaults to the "old" method of handling simultaneous MIDAS file usage.

Under the "old" method of concurrency handling, MIDAS attempts to change the access rights on a MIDAS file (segment directory) from "read only" to "read and write," in order to operate on that file. If another user already has the file open for writing, the attempt to open it for writing will fail. In this event, MIDAS will call RECYCL and try again. This is repeated until the attempt is sucessful or until the number of tries exceeds the maximum number specified in the RECYLA parameter (default = 1ØØØ). The latter triggers a MIDAS error 24. This situation is not uncommon on systems where many users are

attempting to access the same MIDAS file (using this "old" method) and/or the system is heavily-loaded in general. If MIDAS error 24 is occurring with troublesome frequency on your system, try increasing the value of RECYLA -- it may alleviate the problem. RECYLA specifies the number of times that MIDAS should attempt to change the access rights before it gives up. The default setting is 1000, which should be enough unless the system is overloaded or many users are working on the same MIDAS file. If RECYLA isn't large enough, MIDAS may return an error code of 24. Most people will never need to adjust this parameter unless they access a lot of files over the network and are repeatedly getting a MIDAS error 24. Increasing the value of RECYLA can reduce the occurrence of this error.

SHDSEG: Shared Locks

The SHDSEG parameter indicates whether or not the MIDAS "lock," which single-threads the use of MIDAS, is available. As delivered, this parameter is set to .TRUE., indicating that a shared data segment is available for the lock. The next three parameters in KPARAM indicate where the lock is located. Network users can disable this lock; see Section 13.

SLSEG: Shared Lock Segment

The shared lock resides in segment :2020, as defined by the default setting for SLSEG. Unless the user has some vitally important reason for moving the lock to another segment, this parameter should be left alone. Be careful when altering this value as you may wind up with unpleasant side-effects. There is no way to anticipate what users are doing with various segments, so if you move the lock from 2020 to some other location, be sure to edit the command file C_SHAREMIDAS to share the new segment. Also, be careful that this segment isn't being used by someone else for some other purpose.

SLWORD: Shared Lock Table Location

SLWORD, set by default to :177777, is the word number of the shared lock table. If the lock table is moved to another location, be sure the new location is not being used for anything else.

MSEMA1: Semaphore Number

The parameter MSEMA1 is currently set to -16; it specifies the semaphore number for the MIDAS lock. The semaphore is used in establishing a wait list for processes waiting to obtain the MIDAS lock.

STSIZ: Files Open Between Calls

The STSIZ parameter specifies the maximum number of segment subfiles which MIDAS can leave open between calls. It is set to 20 as delivered, meaning that MIDAS can leave at most 20 segment subfiles open between one MIDAS call and the next. Previously, all segment subfiles were closed after each call, significantly eroding performance times. If performance is a problem, try increasing this number -- performance should improve due to less file opening and closing overhead. Be careful, however, of making this number too large, or you will find yourself short of file units! The maximum setting for STSIZ is 128, which is the maximum number of file units available per user. It's obvious that you'd never want to make STSIZ that big anyway.

NOFUNS: Offline Routine File Units

The NOFUNS parameter limits the number of file units simultaneously useable by the offline routines PRIBLD, SECBLD, BILD$R, KBUILD and MPACK. NOFUNS is currently set at 40. Some performance improvement may be had by increasing this number, but again, beware of short-changing yourself on file units by granting too many of them to MIDAS.

NOLVLS: Index Levels

NOLVLS indicates the maximum number of index levels per index subfile that PRIBLD, SECBLD, and BILD$R can attempt to build during a single build operation. The default NOLVLS setting is 18. Based on the index entry size and the block size defined for each level in the index to be built, these utilities estimate and reserve the amount of memory needed to house this index on the assumption that it will have 18 levels. However, this space is only reserved during the index build, and any unused space is always freed up after the index is built.

Appendices

APPENDIX A

MIDAS ERROR MESSAGES

INTRODUCTION

This appendix lists all the MIDAS error codes that the user is likely to encounter during MIDAS file handling. Such errors are returned by internal MIDAS routines, some of which users can call directly. Also explained in this appendix are error messages returned by KX$RFC, an internal MIDAS routine used by the four MIDAS utilities: CREATK, KBUILD, KIDDEL and MPACK.

For information on the error messages returned by KX$CRE, PRIBLD, SECBLD, BILDR, KXTIM and other offline routines, see Section 14.

RUN-TIME ERROR CODES

The following is a list of all MIDAS run-time error codes. Included for each error are: the number of the error, the routines in which each is likely to occur, the cause of the error, and, in some cases, what can be done to recover from or avoid the error. Where appropriate, the COBOL STATUS-CODE equivalent is given. If the COBOL equivalent is missing, the MIDAS error is either not returned to the COBOL user at all, or, as in the case of MIDAS errors 40 and above, the STATUS-CODE equivalent is 99. These error codes are returned directly to the user unless error traps are included in the program. In each interface, MIDAS codes are returned to the user through different mechanisms. For example, in COBOL, condition codes are returned in the STATUS word, as a two-digit code in STATUS-KEY-1 and STATUS-KEY-2. FORTRAN users will recognize that these codes are returned in word 1 of the MIDAS communications array.

Code	Routine	Explanation
1	KX$ELP, KX$NX1	Indicates that duplicates exist for the current key. COBOL STATUS-CODE is 00.
7	KX$NX1, KX$ESH, KX$EDA, KX$ELP, KX$GPT, KX$GNE, KX$ULV	Indicates that the sought-after entry does not exist in the file. COBOL STATUS-CODE is 23 except in INDEXED MIDAS files opened for SEQUENTIAL access: COBOL STATUS-CODE is 10 in this case, and 22 when a REWRITE is attempted without a prior READ.

10	LOCK$, KX$LDR	The data record has the "locked" bit set. COBOL STATUS-CODE is 90.
11	UPDAT$	The data record does not have the "locked" bit set when it should. This happens when an update is attempted without first locking the record. (Method differs in each interface.) COBOL STATUS-CODE is 91.
12	KX$DAD, KX$ADD	Duplicate keys not allowed. COBOL STATUS-CODE is 22 for attempt to add a duplicate primary key. COBOL STATUS-CODE is 92 on attempt to add duplicate secondary key where duplicates are not allowed.
13	KX$CCE	An unrecoverable concurrency error has occurred; for example, the user's current entry was deleted by another user. COBOL STATUS-CODE is 94.
14	FIND$, UPDAT$, NEXT$, DELET$, ADD1$, LOCK$, GDATA$$	MIDAS unable to obtain internal after call to semaphore wait-list manager. Also may indicate internal problems with use of shared lock. This is serious if returned repeatedly on a call. COBOL STATUS-CODE is 94.
20	KX$WPR	Error encountered while writing a record or index block. COBOL STATUS-CODE is 30 for INDEXED files, 96 for RELATIVE files.
21	KX$RPR	Error encountered while reading a data record or index block. COBOL STATUS-CODE is 30 for INDEXED files, 96 for RELATIVE files.
22	KX$OIT	Error on a call to SRCH$$. Check CODE. COBOL STATUS-CODE is 30 for INDEXED files, 96 for RELATIVE files.

23	KX$OIT	Error on a call to SGDR$$. Check CODE. COBOL STATUS-CODE is 30 for INDEXED files, 96 for RELATIVE files.
24	ADD1$	Can't open MIDAS file for reading and writing after trying RECYLA times. (In KPARAM: default setting = 1000.) COBOL STATUS-CODE is 30 for INDEXED files, 96 for RELATIVE files.
25	ADD1$	An error occurred on a call to SRCH$$. Check CODE. COBOL STATUS-CODE is 30 for INDEXED files, 96 for RELATIVE files.
26	KX$MYB	An error occurred on a call to PRWF$$ while trying to read an index block. COBOL STATUS-CODE is 30 for INDEXED files, 96 for RELATIVE files.
27	KX$RAD	An error occurred on a call to PRWF$$ while writing an index block. COBOL STATUS-CODE is 30 for INDEXED files, 96 for RELATIVE files. In RELATIVE files, this happens when writing a RELATIVE key that doesn't match the key as defined in the template.
29	KX$CLS, KX$CIT	An error occurred on a call to SRCH$$ while trying to close a segment. COBOL STATUS-CODE is 30 for INDEXED files, 96 for RELATIVE files.
30	NEXT$, LOCK$, NEXT$$, KX$GET	The user did not ask for the array to be returned when it must. Set FL$RET in flags on the call.
31	UPDAT$	The array must be supplied but was not.
32	KX$DAD, KX$IDE KX$IIE, KX$RIT KX$WIT	User supplied a bad length, for example, a bad data record length. COBOL STATUS-CODE is 95.
33	NEXT$, ADD1$, NEXT$$, KX$ESH, KX$EDA, BILDR, KXULV	The user-supplied array is bad.
34	NEXT$, NEXT$$	Use of NEXT$ is not allowed in direct access files. COBOL STATUS-CODE is 97.

35	KX$ADD	Cannot do an indexed add to a direct access file. COBOL STATUS-CODE is 98.
36	DELET$	The user set FL$USE in flags but the current array involved a different index than the one supplied by the user in this call.
42	KX$LVL	No offspring pointer or no next block found.
44	KX$EDA, KX$REC	Got an index block but expected a data record.
45	KX$GPE, KX$LVL	Got a data record but expected an index block.
46	KX$NXl	Expected duplicate key was not found.
47	KX$DLT	Attempting to delete an entry from an empty block. (KX$DLT deletes index entries.)
49	KX$DLT	Invalid entry number; it could be negative or out of range.
5Ø	KX$DCD	Overflow entries discovered: for old files only. (Time to REMAKE the file!)
51	KX$ELP	Bad index pointer in index entry. For example, segment number is Ø.
52	KX$GIB	Index subfile overflow: too many segments. (Increase index subfile length with EXTEND, then MPACK file; or use MODIFY to make double-length indexes if your version of CREATK permits it. (See L_CREATK in Sections 13 and 15.)
64	CLOS$$	Unable to unlock a data record (the last entry referenced) upon closing the file.
71	KX$DAD	An error occurred on a direct access search. That is, the search was successful when it should not have been.

84	KX$ADD	Error on an attempt to write an index block. The error probably occurred in KX$WBK, which writes the current block (the top entry in the access stack), out to disk. May happen if MIDAS is unable to open or write the segment.
85	KX$ADD	The error actually occurred in KX$IIE. Either the user supplied a bad key length or MIDAS tried to insert an entry into a block that was already full. This means the data subfile is full.
86	KX$ADD	An error occurred in KX$SIB which splits index blocks. Could be any one of many problems.
87	KX$ADD	An error occurred in KX$GNE while attempting to get the next index entry. This could occur if there were no more entries in the index subfile.
88	KX$ADD	An error occurred in KX$GPE (gets previous entry) while attempting to get the preceding entry. The entry may not exist.

Explanation of Error Code Numbers

In case you're wondering why there are "holes" in the numbering sequence, here is a brief explanation:

- Error code numbers 1-13 are non-fatal errors; they will not be printed if your program contains an alternate return. They deal mainly with "find" and update errors which can usually be trapped if you take the time to write good error handlers in the program.

- Error codes 20-28 are disk errors and are generally fatal. They generate error messages that consist of a MIDAS error code plus a file system message. These errors cause the program to abort, returning control to the user. However, they too can be handled in most of the language interfaces to MIDAS by including error traps in your programs.

- Error codes 3Ø-36 are file handler errors, usually generated when arguments are supplied incorrectly to the FORTRAN call level routines like ADD1$ and NEXT$. Some of them (32, 33) deal specifically with inmproper use of the communications array. Check all the calls to the routine in which the error was generated and fix any bad arguments.

- Error codes 4Ø-48 are MIDAS errors which you shouldn't see.

- Error codes 51 and 52 will only appear if you run out of room in a MIDAS file; at this point, you can use the CREATK EXTEND option to increase the segment subfile size. You must then MPACK the file to effect the change. See Section 15 for details.

- Error codes 71-88 are mostly caused by internal MIDAS errors.

ERRORS RETURNED BY UTILITIES

The four MIDAS utilities CREATK, KIDDEL, MPACK and KBUILD all share some common messages which are returned by one MIDAS routine, KX$RFC. This routine checks the file rev stamp to see if it is compatible with the current version of MIDAS. (It's a user-callable routine discussed in Section 14.) These utilities also check to see if the index length defined for the file is compatible with the INDLNT setting in the current version of MIDAS.

KX$RFC Messages

The following messages are returned by KX$RFC:

1. If a file has a major rev. stamp of Rev. 15 or earlier, the message:

   STOP! REMAKE THIS FILE!

   will appear. No further processing will be allowed on this file until the user runs the REMAKE utility. The file will then be compatible with MIDAS versions stamped Rev.16 and above. REMAKE is now in MIDAS>CMDNCØ.

2. If the major rev stamp of the file is greater than that of the version of MIDAS you're currently running, MIDAS would rather not attempt any operation on the file that might endanger its health. The warning message:

   MAJOR REV STAMP OF FILE GREATER THAN THAT OF MIDAS

   is displayed. The current version of MIDAS will work on any file with a major rev stamp lower than its own (no lower than Rev.16) and will automatically update the rev stamp of the

file. However, it is not possible to convert back to an earlier major rev of MIDAS (for example, to go from 18.1 back to 17.4) because the above message will result when you try to access existing MIDAS files. It is possible to convert back and forth between update revs, like 17.4 and 17.6.

3. The INDLNT parameter in KPARAM determines the number of segments which are allocated per index. The default setting is 10. If this number is changed by the user (it's not supposed to be!), problems may arise if a file contains a number different than that of the current version of MIDAS. This can result from a change to INDLNT in a previous version of MIDAS, giving all files created under that version INDLNT number of segments per index. When a new version of MIDAS is installed, with INDLNT set to 10 (by default), problems will arise. The same result occurs if you change INDLNT in the new version of MIDAS, making it different for the INDLNT value contained in existing files. In either case, the following message is returned:

   BASIC INDEX LENGTH OF FILE DOES NOT MATCH THAT OF MIDAS

   If INDLNT has been changed in the current version of MIDAS, change it back to a value compatible with the INDLNT setting in the file or recreate all your old files with the new version of MIDAS, ensuring that they are all built the same way and can be properly used by all the current MIDAS utilities.

4. If the file name passed to a utility is not that of a MIDAS file, KX$RFC returns the error message:

   NOT A VALID MIDAS FILE (caller-name)

   where caller-name is the name of the routine that called KX$RFC. This can happen if:

   - the named file is not a SAM segment directory,
   - if segment subfile 0 (the file descriptor subfile) is not present, or
   - if segment subfile 0 does not contain the proper information to indicate that it is a MIDAS file.

KX$OIT Message

Another message, returned by the KX$OIT routine (which is called by every routine that wants to open an index subfile for update), occurs if KX$OIT is unable to open the file, either because it does not exist or because the use of this index is not supported by the access method being used.

The following message is returned:

```
Segdir unit not open.  (KX$OIT)
UNABLE TO VERIFY/UPDATE MIDAS REV FOR INDEX xx
```

where xx is the number of the index subfile which KX$OIT tried to open.

This can happen in COBOL, for example, if a direct access file is opened for indexed access and the user tries to access the file by secondary key. The program will fail when KX$OIT attempts to open the secondary index subfile because secondary keys (and indexes) are not supported by RELATIVE file structure.

This message can also happen if you are trying to access an index that doesn't exist.

APPENDIX B

MIDAS AND THE FILE SYSTEM

FILE SYSTEM BACKGROUND

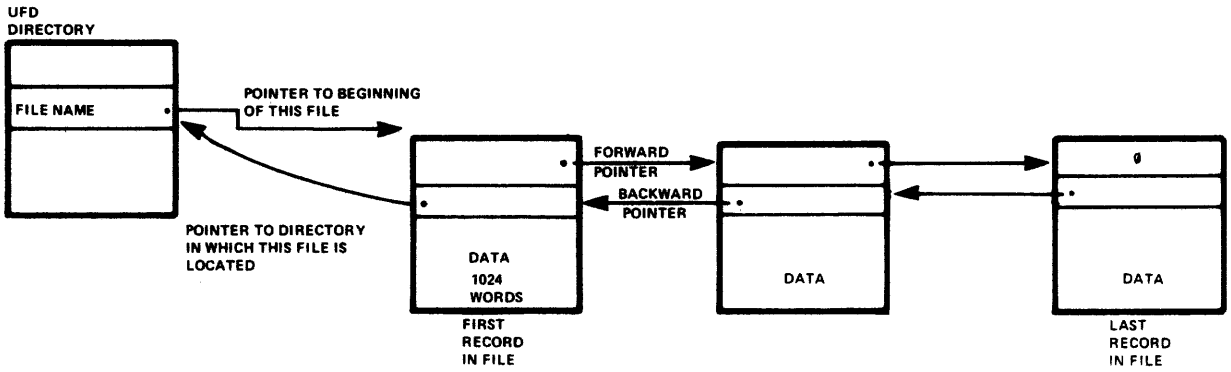
For those users familiar with the PRIMOS file system, this appendix explains how MIDAS file structure is related to file types you already know. Those not familiar with Prime file structure will probably find enough information here to make sense out of the subsequent discussion of MIDAS file structure. See The Subroutine Reference Guide for more information on Prime file structure.

SAM Files

Most files under PRIMOS are structured for sequential access, and are called "sequential access method", or SAM, files. Their records are "strung together" by forward and backward pointers. To get any record in a SAM file, you must step sequentially through the file records until the desired one is found. You cannot randomly retrieve any record in the file; but it is possible to get back to the beginning of the file to reprocess it. From the user's viewpoint, records in SAM file do not have to be the same length; hence they are called "variable-length" records. The file system, however, stores SAM files in 1K word records (1Ø24 words). See Figure B-1 for a picture of SAM file structure.

DAM Files

DAM files are basically SAM files with an extra set of pointers stored in a separate index. DAM files are also called "random" files because records can be accessed randomly in any order. However, to get a particular record, you have to know something about it, like its starting address. From the file system's viewpoint, every record in a DAM file has to be the same length so the starting and ending positions (address) of each are known. This makes random access possible. Figure B-2 shows the logical structure of a simple DAM file. It should be noted that the forward-backward inter-record pointers like those in a SAM file also exist in DAM files so that records can be accessed sequentially also.



SAM File Structure

NOTE:

NON-ZERO INTER-RECORD POINTERS ARE INDICATED BY ARROWS AND DOTS (.); POINTERS WITH ZERO VALUES ARE INDICATED WITH A "Ø" AND DO NOT REFERENCE, OR POINT TO, ANYTHING.

Figure B-1. SAM File Structure

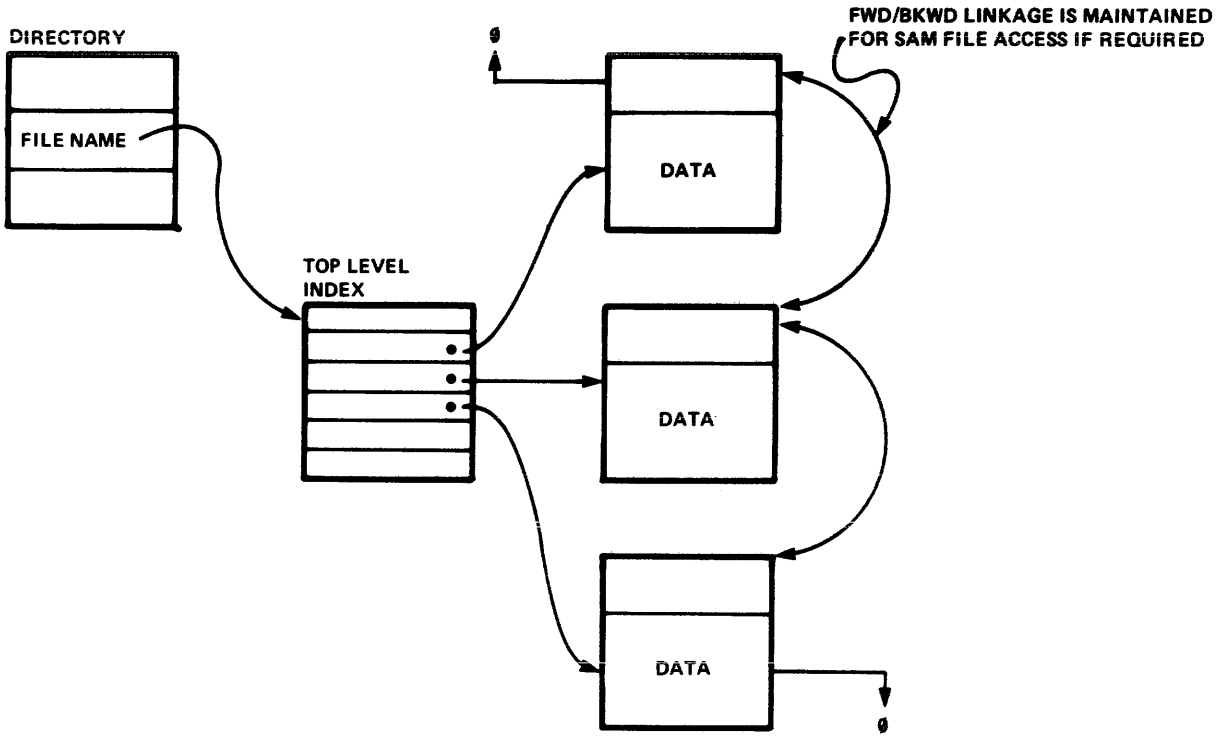


Figure B-2. Simple DAM File Structure

Multi-Level DAM File Structure

The DAM file structure represented in Figure B-2 is a "short" DAM file; it only has a single-level index to the data records in the file. All the individual record pointers are contained in that index level. For larger files, one index level would neither be sufficient nor efficient for quick look-up, and, after all, that's the purpose of direct access files. More index levels have to be created, making one index level an index to the next index level and so on. Figure B-3 shows a three level index structure for a DAM file. MIDAS uses this multi-level index concept in a similar fashion to implement its index subfile structure.

Segment Directories

In PRIMOS terms, a MIDAS file is a special collection of both SAM and DAM files, called a segment directory. A segment directory can be thought of as a network of files, all related and connected by pointers which ultimately reference entries in a data file. Think of it as a large city, like New York, which is divided into several "boroughs" like Manhattan, Brooklyn and the Bronx, but is technically a single city.

A segment directory is a large file similar to a directory (sub-UFD). Its "subordinate" files are not referenced by name as in a sub-UFD, but by number. Refer to Figure B-4 for a picture of a MIDAS file viewed as a segment directory. The various files in a segment directory are called segment subfiles and consist of one or more segments. There are at least four types of segment subfiles in any MIDAS segment directory:

- A file descriptor subfile
- A primary index subfile
- From 1 to 17 secondary index subfiles
- A data subfile

Below is a description of each of these subfiles.

The File Descriptor Subfile: The first segment in a segment directory (which is segment Ø) is the file descriptor subfile. This is nothing more than a list of all the other index subfiles associated with a particular segment directory. It is essentially an overview of the entire segment directory's structure. Any time the structure of an index subfile or a data subfile is changed, the modifications are reflected in the descriptor subfile. This ensures that the description of the file matches the actual state of the file at any given time.

The Data Subfile: The data subfile is the "core" of the MIDAS file.

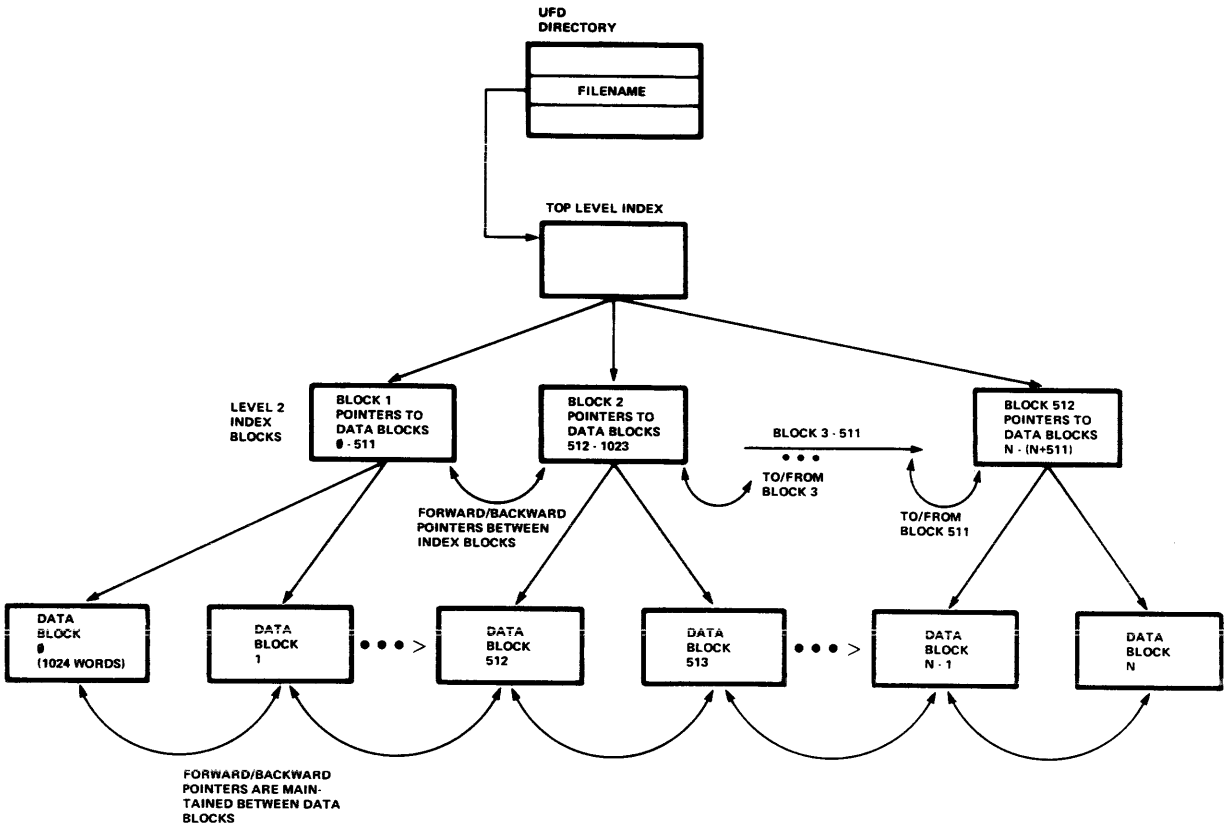


Figure B-3. Multiple Level Index DAM File

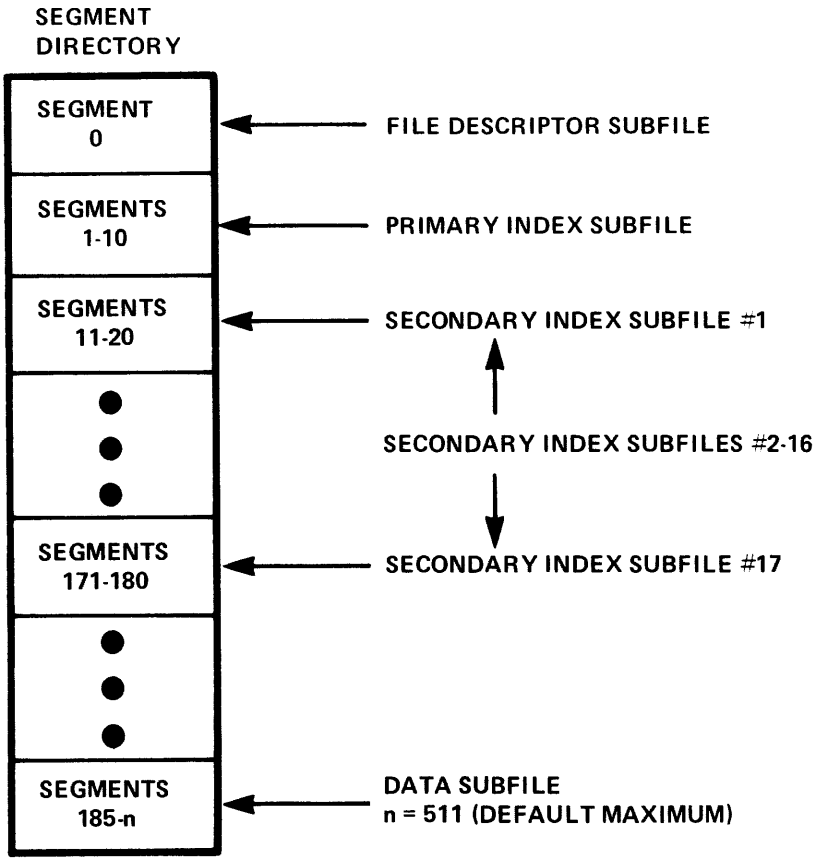


Figure B-4. Segment Directory Structure

It contains the actual data that we want to look up, report on, modify, and so forth. Every record in the data subfile is referenced or pointed to by an entry in one or more of the index subfiles associated with that particular MIDAS file.

The Index Subfile: Every MIDAS file has at least one index subfile, the primary index subfile, and up to 17 secondary index subfiles. Each index subfile consists of one or more segments. The first segment, called an index descriptor block, is a guide to the index subfile and includes the root index block. It tells MIDAS what's in the remaining segments of the subfile. These segments contain index blocks. The number of index blocks per segment depends on the length of the segment; the default value is 512 blocks per segment. However, this can be modified by using the EXTEND option of CREATK. See Section 15.

The Index Block: Each index block contains many entries. Each entry consists of a key-field value (from the data file) plus a three-word pointer to either another index block in the subfile, or to an entry in the data subfile. At the beginning of every index block is information such as key type and length; this information is used in looking up a particular key entry in that index block. Indexes have a hierarchical structure governed by the B-tree algorithm as explained in Section 15.

APPENDIX C

OBSOLETE MATERIAL

INTRODUCTION

This appendix summarizes all the MIDAS routines that are no longer available or are no longer supported. Users of past versions of MIDAS should carefully remove any calls to obsolete routines from their old programs. More information on things that have changed in MIDAS from the previous revision to the present one is available in Appendix D.

OBSOLETE MATERIAL

For your convenience, the obsolete material is divided into three parts:

- Obsolete command files
- Obsolete routines
- Effectively obsolete material

Make sure any obsolete command files or calls to obsolete routines are taken out of all existing user command files and application programs, as they will not work with the new version of MIDAS.

Obsolete Command Files

The following command files have been obsoleted:

- C_FORM: was formerly used to produce an R-mode library with in-line desectorization to help out memory-cramped users with both MIDAS and FORMS running simultaneously. The new interlude alleviates this problem.
- C_NDA4: generated a V-mode library without direct access support; designed for memory-cramped users -- similar to C_FORM. There is no penalty if the user has VKDALB configured for direct access but does not use it. In files that do use direct access, there is a two-word overhead per primary key entry. See Note, below.
- C_NODA: was used to generate an R-mode library without direct access support. With the new V-mode interlude in place, the R-mode library tracks the V-mode library for presence or absence of direct access support making this file unnecessary.

- C_SPCR: generated a special version of CREATK called SPCR. Both C_SPCR and SPCR are now obsolete.
- C_MINIT and C_MSHAR: have been replaced by C_SHAREMIDAS, which shares the MIDAS library, the MIDAS shared lock segment, and intializes everything by running SYSTEM>IMIDAS.
- C_FILL: has been replaced by the C_LIST_MIDAS command file which creates a listing of MIDAS called MIDAS>SOURCE>L_MIDAS.

Note

If users of MIDAS libraries created by either C_NODA or C_NDA4 opt for the default version of the V-mode library, two changes will be apparent:

1. CREATK will ask if the file being created is to be direct access or not.
2. Direct access calls will no longer produce errors.

Obsolete Routines

The following routines are no longer included in MIDAS:

FILSET

KX$BWT

KX$FCL

KX$Ø1X

PRIRAN

SECRAN

SYSINI

The functionality of SECRAN and PRIRAN has essentially been taken over by BILD$R (processes unsorted data) and therefore, it is no longer necessary to keep them alive. Users who previously used PRIBLD, SECBLD and BILD$R will note that FILSET has gone away, meaning that if you haven't already done so, it's time to change all the calls to PRIBLD, SECBLD and BILD$R in existing programs. See Appendix D and Section 14 for details.

Note

The $INSERT file OFFCOM has been removed from MIDAS and should no longer be included in any application programs.

Lesser-Known Obsolete Routines

A few miscellaneous routines have crept into MIDAS over the years which most users have never heard of; however, those users who have incorporated them into their programs (FORTRAN or PMA) will want to be aware of their absence from this version of MIDAS. They are:

- OPEN$: User callable routine formerly used to open a MIDAS segment directory: it should not be confused with OPENM$:, currently used to open a MIDAS file.

- CLOSE$: Obsolete user callable routine used to close a MIDAS segment directory: the companion routine of OPEN$:. Do not confuse it with CLOSM$, which is the routine currently used to close MIDAS files under the new concurrent process handling method.

EFFECTIVELY OBSOLETE MATERIAL

Some command files and routines now considered obsolete or frozen, appear in the current version of MIDAS for those users who have recently been converted from a Rev 15 (or earlier), or for those users who have files created under an ancient version of MIDAS.

Maintained for Compatibility

Below is a list of effectively obsolete routines, still available for use, but not highly recommended for such:

- C_REMAKE: compiles and loads the obsolete utility REMAKE, producing an executable version of REMAKE in MIDAS>CMDNCØ. Since REMAKE comes already built in the MIDAS>CMDNCØ, users won't have much use for this command file. REMAKE should only be used by users converting pre-Rev 15 files to run under a later version of MIDAS. REMAKE used to be a resumable file called *REMAKE.

- C_REVERT: compiles and loads the obsolete REVERT utility, placing it in MIDAS>CMDNCØ. It is unlikely that anyone will be using REVERT except in rare cases where it is necessary to convert files to run on previous versions of MIDAS. REVERT comes already built in MIDAS>CMDNCØ, so there's no need to run this command file unless you delete the existing version of REVERT. Note also that REVERT used to be a resumable file called *REVERT.

- UMODE$: is another routine still present in MIDAS, and formerly used in many applications, which signals single or multi-user mode. It has been obsoleted by the new method of concurrent process handling and should not be used at all under the current version of MIDAS.

Maintained, But Not Enthusiastically

The secondary data feature has been maintained in this version of MIDAS; however, its use is not encouraged. Most people used it to "hide" data in the secondary index, along with a secondary key value. However, if data is this sensitive, perhaps you ought to be using POWER or DBMS to protect it. Although useful if properly implemented, the secondary data feature has generated a lot of confusion among users and needlessly complicates file modification and recovery. At this point, users with secondary data are not being forced to remove it, but its use in new MIDAS files is not condoned.

APPENDIX D

THE MIDAS ARCHIVES

INTRODUCTION

This appendix is meant only for MIDAS users who have used it in the past and want to know what must be changed or deleted in their application programs and command files. New users of MIDAS needn't bother with this unless they are curious about the historical development of MIDAS.

Basically, the information in this section covers these general areas:

- Administrative and installation changes
- Changes in certain methodology for this version of MIDAS
- Changes to utilities and routines that require program modification
- Additions to the repertoire of MIDAS routines and command files

ADMINSTRATIVE/INSTALLATION CHANGES

Installation and administrative modifications involve changes to MIDAS command file names and functions, and functional changes in the IMIDAS and MCLUP utilities.

Command File Changes

Below is a summary, in list form, of the major changes that have occurred to MIDAS in the administrative and installation areas at this revision. Many of these changes will directly impact some user command files, especially if the command files are used routinely in initializing and reloading MIDAS.

- MIDAS>C_MIDAS no longer calls MIDAS>SOURCE>C_REMAKE or MIDAS>SOURCE>C_REVERT. Both MIDAS>CMDNCØ>REMAKE and MIDAS>CMDNCØ>REVERT come already built on the master disk and most users will never need to rebuild them.
- MIDAS>C_INSTALLMIDAS does not copy MIDAS>CMDNCØ>REMAKE or MIDAS>CMDNCO>REMAKE to the system UFD CMDNCØ. However, it does copy MIDAS>CMDNCØ>MPACK to CMDNCØ.

- MIDAS>SOURCE>C_FILL has been changed C_LIST_MIDAS, also in MIDAS>SOURCE. It produces a listing of MIDAS called L_MIDAS in MIDAS>SOURCE.

- MIDAS>SOURCE>C_MPACK now produces MIDAS>CMDNCØ>MPACK rather than MIDAS>SOURCE>*MPACK.

- MIDAS>SOURCE>C_MSHAR and MIDAS>SOURCE>M_INIT have been replaced by MIDAS>C_SHAREMIDAS which does everything the former command files did, plus it runs IMIDAS.

- MIDAS>SOURCE>C_REMAKE now produces MIDAS>CMDNCØ>REMAKE instead of MIDAS>SOURCE>*REMAKE. See Appendix C for more information.

- MIDAS>SOURCE>C_REVERT now produces MIDAS>CMDNCØ>REVERT instead of MIDAS>SOURCE>*REVERT. See also Appendix C.

Changes to Administrative Utilities

Both the IMIDAS and MCLUP utilities have been changed at this revision of MIDAS to improve MIDAS initialization and clean-up.

IMIDAS Changes: IMIDAS initializes the shared lock and semaphore for single-threading MIDAS use. At this revision of MIDAS, IMIDAS can only be run from the system console. If any other user attempts to run it, this message appears:

IMIDAS MUST BE RUN FROM THE SYSTEM CONSOLE (IMIDAS)

Make sure that no one else is using MIDAS when the IMIDAS utility is run, or that person's opened MIDAS files may be corrupted.

MCLUP Changes: Although breaks are inhibited during most of a MIDAS call, it is still possible for a MIDAS user, whether by forced-logout or by breaking out of a program via CTRL/P, to exit MIDAS before the next process on the wait list can be notified that the lock is free. In this case, the user who quits out of MIDAS does release the lock, but simply doesn't get a chance to generate the usual "notify" to the semaphore wait list. The "notify" triggers the next process on the list to attempt to obtain the lock. This doesn't happen often, but when it does, all MIDAS operations come to a halt. All MIDAS users on the semaphore wait list will sit idle, waiting for the notify that never came.

Previously, the only recourse in this situation was to run IMIDAS from the system console. MCLUP, observing that the lock did not contain the user number of a person owning the lock, could do nothing. Now, when the lock is set to zero, indicating that it is free, and the semaphore wait count is negative, indicating one or more users waiting on the semaphore, MCLUP will assume the situation outlined above has occurred and will generate a notify to the semaphore. However, this will only happen if someone remembers to run MCLUP when MIDAS goes out of commission.

After generating the notify, MCLUP will print this message:

Cleanup for unknown user successful

at the terminal of the user who invoked MCLUP.

Changes to KPARAM

The KPARAM file, which contains all the parameters used by MIDAS routines, has been modified at Rev. 17.6 to "fix" several formerly user-modifiable parameters. This means that users who have previously modified the following parameters will be unable to modify them from now on:

- INDLNT
- INDNUM
- MFILES
- MKEYSZ
- NFUNIT
- NOBUFS
- NOIDXS

This should prevent users from making inappropriate changes to these parameters and thus getting into trouble. In most cases, it would not buy users a great deal to modify them anyway, as these parameters have already been set to achieve the maximum level of performance. Parameters which users can currently modify are explained in Section 15.

NEW METHODS AND OLD METHODS

The new version of MIDAS incorporates two new methodologies for handling old problems. These new methods involve concurrent process handling and the elimination of R-mode processing via the new R-mode library interlude.

The R-Mode Interlude

At this revision of MIDAS, the runtime routines in the R-mode MIDAS library, KIDALB, have been replaced by an interlude to the V-mode library. This was done primarily to:

- Reduce memory requirements. Instead of xx copies of the R-mode runtime routines for xx users, there will be just the one shared V-mode MIDAS library copy (VKDALB) and xx copies of the much smaller interlude.

- Speed up I/O time by using all V-mode calls which are much faster than corresponding R-mode calls. Since one MIDAS call typically generates many system calls, the impact should be noticeable.

There is no need to reload R-mode programs using just the MIDAS library if a new copy of the V-mode MIDAS library is brought up. Because the library is dynamically linked with every execution, R-mode programs will need only to be reloaded once to use this new library and never again unless programs are saved after being run -- a rather uncommon practice.

Obviously, this interlude will not work on a P300.

Related Changes: The interlude brings with it several other changes which have been mentioned briefly in other sections, but which are gathered here to provide an overall summary:

- The R-mode library will now track the V-mode library both in the presence or absence of direct access support.

- ERROPN, FILERR, FILHER, and KX$TIM will be the only internal MIDAS routines callable by V-mode and R-mode users. Previously, V-mode users had access to no internal routines at all. Since the R-mode library tracks the V-mode library in functionality, calls to internal routines have to be restricted. Several routines have been dropped because of this, including FILSET, KX$BWT, KX$FCL, KX$OIX, PRIRAN, SECRAN, and SYSINI which have all been obsoleted or duplicated in functionality elsewhere in MIDAS.

Concurrent Process Handling

The new method of concurrent process handling involves new open and a close routines, a shared lock and semaphore wait list which single-thread MIDAS use. The current method improves MIDAS performance significantly by reducing file system overhead. It may be helpful to summarize how MIDAS previously handled concurrent access before explaining how the new method works.

Previous Concurrency-Handling Methods: In previous releases, MIDAS coordinated concurrent processes by gating processes at the segment subfile (MIDAS file index) level. This method relied upon file system read/write locks, which were previously set at 2, and required that segment subfiles be opened at the start of each MIDAS file operation and be closed upon completion of that operation.

With this method, only one process could access a file segment at a time. A second process could only proceed after the first process finished its MIDAS operation and the file segments were closed. This method of synchronization required many calls to the file system routine SRCH$$ to open and close file segments, thus imposing a significant performance penalty.

For example, to retrieve a record, MIDAS opened the index segment subfile(s) and the data segment subfile. When the retrieval was completed, MIDAS closed these segment subfiles. This was repeated for every operation on a MIDAS file, resulting in substantial disk I/O overhead during sequential processing of large files.

The New Method

MIDAS's present method of handling concurrent processes improves performance by greatly reducing the number of file system calls. By using a semaphore and a "lock" in shared memory, MIDAS simply allows only one process at a time to execute a MIDAS file operation. Therefore, MIDAS segment subfiles need not be closed at the end of each operation only to be reopened at the start of the next call, as was the case in the old method. See Section 13 for details.

For FORTRAN/PMA Users Only: This method also requires that the user open and close MIDAS files with one of two methods, described in Section 6. These methods use either the OPENM$ and CLOSM$ routines, or the NTFYM$ routine, all of which are recent additions to the MIDAS repertoire. Section 6 tells you how to use them. See also NEW ROUTINES below.

CHANGES TO EXISTING ROUTINES

Some routines have been rewritten and/or have had their calling sequences changed. Make sure you make the appropriate adjustments in your programs.

Changes to Offline Routines

The offline file building routines PRIBLD, SECBLD, and BILD$R have been rewritten for 17.6 in V-mode. This was done to make better use of file units by leaving them open between calls, reduce disk usage through a different index build algorithm, and generate more useful diagnostics. User application programs calling these routines will have to be

changed to agree with the new calling sequences, taking care also to delete $INSERT SYSCOM>OFFCOM statements from all programs because OFFCOM no longer exists. You should also delete calls to FILSET, KX$BWT, KX$FCL (which are all obsolete), and any calls to the file system to open the MIDAS file descriptor subfile on some file unit specified by ISUNIT. ISUNIT no longer exists either. The jtemps argument, a three-word array formerly used in all calls to these offline routines has been replaced by an INT*2 sequence flag. Details on the sequence flag can be found in Section 14.

Calls to PRIBLD, SECBLD and BILD$R may still be made from the same program to build different indexes at the same time, but these indexes must all belong to the same MIDAS file. Indexes from different MIDAS files cannot be built by these routines by the same program at the same time.

PRIBLD, SECBLD and BILD$R are now able to build both output files with variable-length data records and direct access MIDAS files. See Section 14 for the new calling sequences for these routines and a list of the new error messages they generate. It should be noted that when one of these routines returns an error message other than the ones listed in Section 14, it is almost guaranteed that the file being built will be useless. In this case, it is suggested that the user zero the file (use KIDDEL), try to figure out what went wrong and try again.

Changes to FORTRAN Subroutines

No new MIDAS subroutine calls have been added to the FORTRAN/PMA interface, but the file-no parameter, present in the general calling sequence as represented in the old MIDAS manual, has been obsoleted. See the new calling sequence in Section 6. Also an addition to the MIDAS flags parameter, which some users may already know about, was made, allowing users access to the index entry immediately prior to the current one. This is done by setting bit 11 in flags, called FL$PRE. See Section 6 for further information.

Changes to Utilities

All of the MIDAS utilities have been modified for this version of MIDAS. Some of the dialogs have changed significantly, producing different prompts and sometimes in new sequences; others have changed internally, with little impact on the responses required of the user. The utilities that have been changed in a major user-visible way are discussed here, with additional information available in other sections of this book as indicated. Users should read this information carefully and make the appropriate changes to any command files that invoke these utilities.

Changes to CREATK

CREATK and all of its subroutines have been rewritten. CREATK must now run in in V-mode. User input to prompts is checked as thoroughly as possible, and error messages, where they appear, are considerably more informative than in previous versions.

Summary of Enchancements and Alterations: Although the rewrite of CREATK attempted to maintain a high degree of compatibility (for the sake of veteran users with a plethora of command files), some unavoidably incompatible changes were forced by the adoption of B-trees at rev 16.5. The list below summarizes all of CREATK's major user-visible changes, most of which were not specifically addressed in Section 2 to spare new users additional trauma.

- Lowercase input is now accepted; in fact, a mixture of upper- and lowercase input is fine.

- Questions requiring a yes/no type of response will not accept a carriage return or blank line as a valid reponse. Such questions will accept "YES", "NO", "AYE", "NAY", or "OK" in upper, lower, or mixed cases.

- Treenames (also called "pathnames") are now accepted in place of filenames, if this has not been stated previously.

- Hard-wired file units have been eliminated.

- When creating a new file, CREATK now asks for "PRIMARY KEY TYPE" and "PRIMARY KEY SIZE" instead of "KEY TYPE " and "KEY SIZE ." The messages for secondary indexes remain unchanged. This should make things clearer for new users.

- The key size scanner is now more lenient and does better key size checking. The maximum key size is 16 words (32 bytes or 256 bits) for bit strings, and 32 words (64 bytes) for ASCII strings. Integer and floating-point key types remain predefined by the hardware.

- If the file, whether a new one under creation or an old one being modified, is enabled for direct access, the number of records specified by the user to be pre-allocated is checked to make sure that: (1) it is greater than Ø.Ø, (2) it is a whole number and (3) it can actually be pre-allocated given the file's maximum segment directory size and maximum number of words per segment.

- Secondary index numbers greater than 17 are not accepted. If secondary indexes 18, 19 and 2Ø existed, they would overlay part of the data segment subfiles.

- If you are using a version of CREATK that allows double-length indexes (i.e., one that was created by L_CREATK), the new version of CREATK will not allow index number 17 to be double length. This, too, would overlay part of the data segment subfiles.

- The user provided secondary data size is checked to make sure that at least two index entries can be inserted in a last level index block. In case of an error, the maximum secondary data size for that index is printed, given the maximum index block size defined by the parameter RECLNT in MIDAS>SOURCE>KPARAM, normally set at 1Ø24 words.

- The user-specified block size for the extended options path is checked to ensure that at least two index entries can be inserted per index block on that level. On error, the minimum block size for that index at that level is printed. The maximum is assumed to be defined by the parameter RECLNT in MIDAS>SOURCE>KPARAM, normally 1Ø24 words.

- The key size is no longer requested at each index block level in the extended options path. With the advent of B-trees, the index key size is assumed to be uniform through all levels of an index B-tree.

- Previously, if running a version of CREATK with double-length index support enabled, the query "DOUBLE LENGTH INDEX?" was asked before the primary key type and key size were requested, and after the key type and key size for each secondary index were requested. CREATK now uniformly queries the user after the key type and size have been entered, regardless of whether the primary or secondary index is being defined.

- The file read/write lock is set to n readers and n writers on file creation by default. CREATK informs the user of this by displaying the message "SETTING FILE LOCK TO N READERS AND N WRITERS" at the end of each file creation session. You can have CREATK set the file read/write locks to the system default setting, usually n readers and one writer, by substituting the key M$NR1W for M$NRNW in the call to KX$CRE in CREATK (MAIN).

- A new file is not created until all the questions necessary to define a template have been answered properly. A user may therefore break (CTRL/P) out of CREATK any time before this and not end up with a partially created file.

- If a file system error is encountered, CREATK responds in a reasonably polite manner, telling you what the file system error is instead of just giving you a MIDAS error code. If the error occurs while creating a new file, CREATK, through KX$CRE, attempts to delete the partially created file and reports the original file error, ignoring any errors it may have encountered while attempting to delete it. If the file system error occurs while processing an already existing file, the file(s) are just

closed and the original error is reported with errors encountered in closing ignored.

- When you ask to work on an already existing file, MIDAS examines it carefully to see that it is indeed a valid MIDAS file. The routine KX$RFC, documented in Section 14, is called to do this verification. See Appendix A for messages commonly returned by KX$RFC.

Changes to CREATK Options: The following changes apply specifically to the various options of CREATK that have been changed in this version of MIDAS.

- The EXTEND option now checks the user-specified segment directory size to ensure that it is not smaller than the current data file growth point. If it is, the message "ALREADY USED THROUGH SEGMENT nnn" is displayed, where nnn is the data file growth point, and the question is re-issued. As before, a Ø response to either question of the extend options dialog causes CREATK to assume the default sizes. The minimum segment subfile size is still 64K words. See Section 15 also.

- The LUSAGE command is no longer displayed as a valid function by the HELP command. The LUSAGE command is still supported, that is, you can still use it, but it does the same thing as the USAGE command. Previously, the only difference between the two was that LUSAGE also printed the version of MIDAS that last modified the file. Both commands now do this and LUSAGE is being de-emphasized to avoid confusion.

- The MODIFY function no longer exits to Primos upon receipt of a carriage return or blank line for an index number. Instead, it will return to the question "FUNCTION?", making it consistent with the behavior of the other functions.

- When MODIFYing a double-length index, CREATK will first check to see whether the index already contains enough entries to to cause it to overflow a normal index. This means it would now be using the additional space available in a double-length index. If so, MODIFY will not ask whether the index should remain a double-length one or not; it is assumed that the index must remain a double-length index, preventing the user from trying to create a new index on top of the already existing long index. MODIFY will ask if an index is to remain a double-length index, regardless of whether your version of CREATK supports double-length indexes or not, only if the index already is double-length and the index is not using its extra space yet. MODIFY will not ask if an index is to become double-length if the index following it already exists, or if the index in question is secondary index 17, or if the running version of CREATK is not enabled to ask about double-length indexes.

- The PRINT and SIZE functions always print the key size in words and bits or bytes now. Previously, if the key type was integer or floating point, the key size was printed in words only.

- The PRINT function, when giving statistics about index "DATA"" (the data subfile), will print "PRIMARY KEY SIZE" rather than "KEY SIZE" to distiguish which index data key size is being printed. Users are reminded that the number of entries printed for index "DATA" are the number of entries as of the last MPACK of the file.

- The VERSION function does not make reference to overflow blocks or overflow chains anymore. The whole concept of overflow was eliminated with the advent of B-trees at rev 16.5.

Changes to KIDDEL

KIDDEL has been rewritten to run in V-mode, and now accepts the use of pathnames, does not use hard-wired file units, and can zero (initialize) as well as delete the various parts of a MIDAS file.

Summary of Changes: Instead of asking the single question:

INDICIES?

KIDDEL now asks the following question:

DELETE INDEXES:

The user may answer with one of the following responses:

- A list of index numbers to be deleted -- this deletes just certain index subfiles

- ALL -- which deletes entire file

- JUNK -- which gets rids of useless segment subfiles created as a result of an aborted MPACK operation

- NONE - which causes KIDDEL to display the "ZERO INDEXES" prompt: see below

If you supply a list of valid index numbers to be deleted, optionally separated by commas, only these indexes will be deleted. KIDDEL then exits to PRIMOS. The space occupied by these segement subfiles will be reclaimed, and the deleted indexes will no longer exist. If you want them back, re-add them with the ADD option of CREATK (See Section 15). The numbers 18, 19, and 20 are not accepted as valid secondary index numbers. An index number of Ø, representing the primary index, is not accepted either, for to delete the primary index implies deleting the entire file. Use the ALL option to delete the entire file.

The JUNK option may be specified by itself or with the list of indexes. It will cause any useless segment subfiles created by an aborted MPACK operation to be deleted. KIDDEL will then exit to PRIMOS.

ALL must be entered by itself and causes the entire MIDAS file to be deleted from the user's UFD. Again, KIDDEL will then exit to PRIMOS.

The NONE option initiates the other KIDDEL dialog -- the one that lets you initialize various parts of a MIDAS file so you can re-use part or all of the template. A NONE response is not the same thing as just a carriage return or blank line, both of which cause the question to be re-asked.

The "zero indexes" dialog begins with this question:

ZERO INDEXES:

Enter one of the following in response to this prompt:

- A list of indexes to be zeroed -- the same rules as above apply. Indexes 18, 19, and 20 are not valid index numbers, and if you want to zero index 0, the primary index, use the "ALL" option.

- ALL -- will cause the entire MIDAS file consisting of the the primary index, the secondary index, and the pre-allocated records of a direct access file to be zeroed. Disk space is recovered by truncating the index levels back to just a root block, and deleting the resultant empty segment subfiles. Empty data segment subfiles are also deleted.

- NONE -- will cause KIDDEL to exit to PRIMOS without doing anything to the file.

Note that the "zero" operation will preserve the file read/write lock and non-standard segment subfile and segment directory lengths. In fact, if the user has specified the non-standard lengths with CREATK, but has yet to put them into effect by running MPACK, KIDDEL will effectively "MPACK" the indexes that is zeroes. Unlike previous versions of KIDDEL, the current version will not continue if the user inputs a blank line or index numbers that don't exist, but instead will re-ask the last question until a suitable response is received. KIDDEL, like all the other major utilities, calls KX$RFC to verify that the user-specified filename is really that of a MIDAS file.

Changes to KBUILD

The KBUILD utility has been re-written to provide for more flexibility, greater speed and accuracy and better error-checking. KBUILD's dialog has changed slightly, so check out the dialog explanation in Section 3 before using the new version. Other changes made to KBUILD are:

- KBUILD now supports binary files written by FORTRAN with the routine O$BDØ7, under the new "FTNBIN" file type.
- Packed decimal keys, and primary keys that don't start in the beginning of the data record are supported under the RPG file type.
- The "COBOL" file type accepted by KBUILD now works with uncompressed files: the "TEXT" file type should be used with compressed files. (Files that have been edited by the PRIMOS Editor (ED) are compressed.)
- KBUILD now builds MIDAS files with variable-length records. The output file must have had variable-length records specified for it during CREATK.
- When the user specifies an input file or files as being sorted, checking is now done to ensure that KBUILD will not attempt to build an index that already contains entries.
- KBUILD will process secondary indexes that contain legal duplicate key values.
- The error/logging milestones produced by KBUILD have been altered to deal with times that cross midnight.

Changes to Internal Routines

The routines affected by changes at Rev 17.6 are ERROPN and KX$TIM. Although minor, the changes do require the user to alter existing programs that call them.

Changes to ERROPN

In previous version of MIDAS, if a blank line was supplied for a requested filename, ERROPN would fail. ERROPN now accepts a blank line as meaning no file is to be opened and thus informs the caller by returning an funit of Ø. Another change to ERROPN is that the caller now receives the file unit that the error/logging file is opened on in the funit parameter, as opposed to supplying an actual file unit number on which to open the error/logging file, as was the case in the past.

Changes to KX$TIM

The changes to KX$TIM involve both its function and its calling sequence.

- KX$TIM now uses ERRFIL (in common area) as set up by ERROPN to figure out where the error logging file is opened. Previously, there was a fourth argument located between what is now the first and second arguments that specified the file unit on which the error logging file was opened.

- The numrec parameter, which indicates the number of records to be processed for a particular milestone, has a new "special case" value. If set to -1, a milestone of Ø with no header and without intialization, can be generated.

- The counters have been enlarged and the computation of the elapsed time has been fixed to take into account a milestone interval that crosses midnight.

See Section 14 for KX$TIM's calling sequence and an explantion of its purpose.

Additions to PARM.K

The MIDAS parameter file that contains all the keys, variables etc. used by the FORTRAN subroutines in MIDAS is PARM.K. It is copied to the system UFD SYSCOM when MIDAS is installed. There are several new parameters in this file as of Rev. 17.6. They are: M$DACC, M$NRNW, M$NR1W, M$DLNG, M$DUPP, M$BIT, M$BYTE, M$WORD, M$BSTR, M$SPFP, M$DPFP, M$SINT, M$LINT, M$ASTR, ME$NMF, ME$BAS, ME$BDS, ME$BKS, ME$BKT, ME$BL1, ME$BL2, ME$BLL, ME$CBD, and ME$NDA. They are used mainly by the offline routines documented in Section 14.

NEW MIDAS ROUTINES

The new version of MIDAS contains several routines never before made available to users. Two of these will be of interest only to people that like using offline routines rather than the standard MIDAS utilities to do their template creation and file building. The others will be of immediate interest to all FORTRAN and PMA users, as they are integral to the operation of the concurrency handler.

New Internal Routines

At rev 17.6, two routines KX$CRE and KX$RFC have been added that allow users to create and read the configuration of MIDAS files from programs. KX$RFC is called by most of the utilities to find out

whether the file is compatible with the current version of MIDAS, and to discern its configuration. KX$CRE is actually part of the CREATK utility which is called by that utility to set up a MIDAS file template. Both of these routines are documented in Section 14.

New File I/O Routines

FORTRAN and PMA users will need to use either OPENM$ and CLOSM$ to open and close their MIDAS files, or they can keep their old programs as is as long as they insert calls to NTFYM$ before and after they process a MIDAS file. These three routines were released at Rev 17.2, but some users may not have found out about them yet. See Section 6 for details on how to use them.

New Install File

The C_INSTALLMIDAS command file is new at Rev 17.6. It installs MIDAS by copying various files and command files to the UFD SYSTEM. It also copies the MIDAS libraries to the system UFD LIB, and copies all the parameter files to the system UFD SYSCOM, and copies all the MIDAS utilities (except REVERT and REMAKE) from MIDAS>CMDNCØ to the system UFD CMDNCØ.

Index