MIDASPLUS™ User's Guide

Revision 22.0

DOC9244-2LA

MIDASPLUS™ User's Guide

Second Edition

by

Andrew Munro

This guide documents the software operation of the Prime Computer and its supporting systems and utilities as implemented at Master Disk Revision Level 22.0 (Rev. 22.0).

Prime Computer, Inc.
Prime Park
Natick, Massachusetts 01760

The information in this document is subject to change without notice and should not be construed as a commitment by Prime Computer, Inc. Prime Computer, Inc., assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

Copyright © 1988 by Prime Computer, Inc. All rights reserved.

PRIME, PR1ME, PRIMOS, and the PRIME logo are registered trademarks of Prime Computer, Inc. DISCOVER, INFO/BASIC, INFORM, MIDAS, MIDASPLUS, PERFORM, PRIMAN, Prime INFORMATION, Prime INFORMATION/pc, PRIME/SNA, PRIMELINK, PRIMENET, PRIMEWAY, PRIMIX, PRISAM, PST 100, PT25, PT45, PT65, PT200, PT250, PW153, PW200, PW250, RINGNET, SIMPLE, 50 Series, 750, 850, 2250, 2350, 2450, 2455, 2550, 2655, 2755, 4050, 4150, 4450, 6150, 6350, 6550, 9650, 9655, 9750, 9755, 9950, 9955, and 9955II are trademarks of Prime Computer, Inc.

PRINTING HISTORY

First Edition (DOC92444-1LA) July 1985 for Rev. 19.4
Update 1 (UPD9244-11A) January 1986 for Rev. 20.0
Update 2 (UPD9244-12A) August 1986 for Rev. 20.2
Update 3 (UPD9244-13A) July 1987 for Rev. 21.0
Second Edition (DOC9244-2LA) October 1988 for Rev. 22.0

CREDITS

Editorial: Thelma Henner, Eric Wurzbacher
Project Support: Judy Paris
Illustration: Therese Bacharz
Document Preparation: Jeff Cohen
Production: Jean Fitzgerald

HOW TO ORDER TECHNICAL DOCUMENTS

To order copies of documents, or to obtain a catalog and price list:

United States Customers	International
Call Prime Telemarketing, toll free, at 1-800-343-2533, Monday through Friday, 8:30 a.m. to 5:00 p.m. (EST).	Contact your local Prime subsidiary or distributor.

CUSTOMER SUPPORT

Prime provides the following toll-free numbers for customers in the United States needing service:

1-800-322-2838 (within Massachusetts)	1-800-541-8888 (within Alaska)
1-800-343-2320 (within other states)	1-800-651-1313 (within Hawaii)

For other locations, contact your Prime representative.

SURVEYS AND CORRESPONDENCE

Please comment on this manual using the Reader Response Form provided in the back of this book. Address any additional comments on this or other Prime documents to:

Technical Publications Department
Prime Computer, Inc.
500 Old Connecticut Path
Framingham, MA 01701

Contents

ABOUT THIS BOOK xi

PART I -- OVERVIEW

1 INTRODUCTION

Accessing MIDASPLUS 1-2
MIDASPLUS Terms and Concepts 1-2
MIDASPLUS File Access Methods 1-4
Execute-only MIDASPLUS 1-4
Language Groups 1-5

PART II -- FILE CONSTRUCTION

2 CREATING A MIDASPLUS FILE

Dialogs 2-1
File Read/Write Locks 2-3
Sample File 2-4
Variable-length Records and Space Usage 2-5
Keyed Access Dialog (Minimum Options) 2-6
Direct Access Dialog (Minimum Options) 2-9
Optional CREATK Features 2-11

3 BUILDING A MIDASPLUS FILE

Input Files 3-2
Building a Variable-length File 3-6
Adding Secondary Index Entries Only 3-7
Error Reporting 3-8
The KBUILD Dialog 3-9
KBUILD Examples 3-11
Alternatives to KBUILD 3-22
KBUILD Error Messages 3-22

PART III -- FILE ACCESS

4 INTRODUCTION TO FILE ACCESS

Access Operations 4-1
Language Access 4-2
Direct Access 4-2
Running MIDASPLUS on PRIMIX 4-2

5 THE FORTRAN INTERFACE

The Current Record 5-1
Direct Access in FORTRAN 5-2
The Communications Array 5-2
$INSERT Mnemonics 5-4
MIDASPLUS Flags 5-4
Compile and Load Sequence 5-9
The FORTRAN/MIDASPLUS Interface Subroutines 5-10
Opening and Closing MIDASPLUS Files 5-12
OPENM$ 5-13
CLOSM$ 5-15
NTFYM$ 5-16
ADD1$ 5-17
Reading a MIDASPLUS File 5-25
FIND$ 5-25
NEXT$ 5-31
GDATA$ 5-35
Updating a Record 5-37
LOCK$ 5-37
UPDAT$ 5-41
DELET$ 5-44
FORTRAN Programming Example 5-47

6 THE COBOL INTERFACE

Language Dependencies 6-2
Summary of COBOL Statements 6-4
Defining an Indexed MIDASPLUS File 6-4
Error Handling 6-11
File Position 6-14
Reading a File 6-17
Adding Records 6-21
Updating Records (REWRITE) 6-22
Deleting Records 6-22
Indexed Programming Example 6-24
Direct Access Files in COBOL 6-28
Accessing RELATIVE Files 6-30
RELATIVE Programming Example 6-35

7 THE BASIC/VM INTERFACE

Language Dependencies 7-1
Summary of Access Statements 7-2
Locking and Unlocking Records 7-3
Opening/Closing a MIDASPLUS File 7-3
Error Handling 7-4
File Positioning 7-5
The REWIND Statement 7-6
Adding Records 7-7
Reading Records 7-9
Updating Records 7-12
Deleting Records 7-13

8 THE PL/I INTERFACE

Running a PL/I Program 8-2
Opening/Creating a MIDASPLUS File 8-3
File I/O Concepts in PL/I 8-5
Adding Records 8-6
Reading a MIDASPLUS File 8-9
Updating File Records 8-13
Deleting Records 8-15
Accessing CREATK-defined Files 8-17
Error Handling 8-18

9 THE VRPG INTERFACE

Language-dependent Features 9-2
Compile and Load Sequence 9-2
Describing a MIDASPLUS File in VRPG 9-3
File Operations 9-8
Indexed File Examples 9-13
Direct Access in VRPG 9-20
Multiple Key Processing 9-20
Processing with Secondary Keys 9-25
Alternate File Processing 9-30

PART IV -- MAINTENANCE AND ADMINISTRATION

10 THE MDUMP UTILITY

MDUMP Options 10-1
The Sequential Dump File 10-3
The MDUMP Dialog 10-4
Status and Descriptive Messages 10-5
Error Messages 10-8
Sample MDUMP Session 10-8

11 DELETING A MIDASPLUS FILE

The KIDDEL Utility 11-1
KIDDEL Dialog 11-2
KIDDEL Error Messages 11-3

12 CLEANING UP A MIDASPLUS FILE

MPLUSCLUP Options 12-2
Remote Cleanup 12-2

13 MONITORING A MIDASPLUS FILE

User Interface 13-1
Record Locks Display 13-2
Statistics Display 13-3
Configuration Display 13-6
Keys of Locked Records Display 13-12
Errors 13-13

14 ADDITIONAL CREATK FUNCTIONS

Function Summary 14-1
Examining a File 14-2
Modifying a Template 14-7
The Extended Options Path 14-14
Extended Options Dialog 14-15

15 PACKING A MIDASPLUS FILE

Functions and Options of MPACK 15-1
MPACK Dialog 15-4
Abnormal Termination of MPACK 15-5
MPACK Error Messages 15-10

16 INSTALLING AND ADMINISTERING MIDASPLUS

Installing MIDASPLUS the First Time 16-1
Upgrading MIDASPLUS 16-2
Sharing MIDASPLUS 16-2
MIDASPLUS Components 16-3
Providing Access to MIDASPLUS 16-3
Initializing MIDASPLUS 16-4
MSGCTL 16-9
Networking MIDASPLUS 16-10
System Error Logging 16-11

PART V -- OFFLINE ROUTINES

17 OFFLINE CREATE ROUTINES

KX$CRE 17-1
KX$RFC 17-6

18 OFFLINE BUILD ROUTINES

Guidelines 18-2
Restrictions 18-3
Event Sequence Flag 18-3
PRIBLD 18-5
SECBLD 18-7
BILD$R 18-8
Offline Routine Example 18-9
PRIBLD, SECBLD, and BILD$R Error Messages 18-14

PART VI -- APPENDICES

A GLOSSARY A-1

B ERROR MESSAGES

KBUILD Error Messages B-1
MDUMP Status and Descriptive Messages B-3
MDUMP Error Messages B-5
KIDDEL Error Messages B-5
SPY Errors B-5
MPACK Error Messages B-6
KXCRE Error Messages B-7
PRIBLD, SECBLD, and BILD$R Error Messages B-8
Runtime Error Codes B-12
COBOL Status Codes B-16

C PRIMOS ERROR MESSAGES C-1

D USING PRIME CUSTOMER SERVICE D-1

E CONCURRENCY ISSUES

Locked Records E-1
Deleted Records E-4
COBOL Sequential Access E-5
Hard-coded File Units E-6
Concurrency Rules E-7

F OTHER MIDASPLUS OFFLINE ROUTINES

ERROPN F-1
KX$TIM F-2

G THE CALL INTERFACE WITH C

Callable Interface Example G-2

H THE CALL INTERFACE WITH PASCAL

Callable Interface Example H-2

I FILE UNIT MANAGEMENT
MIDASPLUS File Unit Utilization I-1
Potential Problems I-6

INDEX X-1

About This Book

This book is a user guide to MIDASPLUS™, Prime's Enhanced Multiple Index Data Access System. The book is organized in six parts, including eighteen chapters, nine appendixes, and an index.

Part I, containing Chapter 1, introduces MIDASPLUS, discusses MIDASPLUS terms and concepts, provides an overview of MIDASPLUS file access methods, and summarizes the MIDASPLUS language interfaces.

Part II explains how to create and build MIDASPLUS files using the CREATK and KBUILD utilities. Part II contains Chapter 2, CREATING A MIDASPLUS FILE, and Chapter 3, BUILDING A MIDASPLUS FILE.

Part III discusses how to use the MIDASPLUS language interfaces. Part III contains Chapter 4, INTRODUCTION TO FILE ACCESS, Chapter 5, THE FORTRAN INTERFACE, Chapter 6, THE COBOL INTERFACE, Chapter 7, THE BASIC/VM INTERFACE, Chapter 8, THE PL/I INTERFACE, and Chapter 9, THE VRPG INTERFACE.

Part IV explains how to perform maintenance on MIDASPLUS. Part IV contains Chapter 10, The MDUMP UTILITY, Chapter 11, DELETING A MIDASPLUS FILE, Chapter 12, CLEANING UP A MIDASPLUS FILE, Chapter 13, MONITORING A MIDASPLUS FILE, Chapter 14, ADDITIONAL CREATK FUNCTIONS, Chapter 15, PACKING A MIDASPLUS FILE, and Chapter 16, INSTALLING AND ADMINISTERING MIDASPLUS.

Part V explains how to create and build MIDASPLUS files using offline routines. Part V contains Chapter 17, OFFLINE CREATE ROUTINES, and Chapter 18, OFFLINE BUILD ROUTINES.

The nine appendixes are Appendix A, GLOSSARY, Appendix B, ERROR MESSAGES, Appendix C, PRIMOS ERROR MESSAGES, Appendix D, USING PRIME CUSTOMER SERVICE, Appendix E, CONCURRENCY ISSUES, Appendix F, OTHER MIDASPLUS OFFLINE ROUTINES, Appendix G, THE CALL INTERFACE WITH C, Appendix H, THE CALL INTERFACE WITH PASCAL, and Appendix I, FILE UNIT MANAGEMENT.

NEW FEATURES

This book includes details on the following features that are new at Rev. 22.0:

- The ability to set minimum and maximum size limits on Variable-Length Record (VLR) files. (See Chapter 2.)
- A new insert file for COBOL programmers who want to refer to error codes and key values by mnemonic names instead of absolute values. (See Chapter 5.)
- A new SPY option that shows who is holding a locked record in a local ASCII file; the primary key value and the user number are shown. (See Chapter 13.)

Also, Chapter 6 details the COBOL support for variable-length records that was added at Rev. 20.2.

ADDITIONAL DOCUMENTATION

The following Prime publications contain additional information useful in conjunction with this book:

PRIMOS User's Guide, Rev. 22.0	DOC4130-5LA
BASIC/VM Programmer's Guide, Rev. 17.2	FDR3058-101A
Update pages, Rev. 18.1	COR3058-001
Update pages, Rev. 19.0	COR3058-002
Update pages, Rev. 19.4	UPD3058-33A
COBOL 74 Reference Guide, Rev. 20.0	DOC5039-2LA
Update pages, Rev. 20.2	UPD5039-21A
Update pages, Rev. 21.0	UPD5039-22A
FORTRAN Reference Guide, Rev. 17.2	FDR3057-101A
Update pages, Rev. 18.1	COR3057-001
Update pages, Rev. 19.0	COR3057-002
Update pages, Rev. 19.4	UPD3057-33A
Update pages, Rev. 21.0	UPD3057-34A

FORTRAN 77 Reference Guide, Rev. 19.4	DOC4029-4LA
Update pages, Rev. 20.2	UPD4029-41A
Update pages, Rev. 21.0	UPD4029-42A
PL/I Reference Guide, Rev. 19.4	DOC5041-1LA
Update pages, Rev. 21	UPD5041-11A
RPG II V-Mode Compiler Reference Guide, Rev. 20.0	DOC5040-2LA
Update pages, Rev. 21.0	UPD5040-22A

OTHER HELPFUL DOCUMENTS

The Guide to Prime User Documents (DOC6138-5PA) provides a brief description of each of Prime's technical documents.

Release Updates provide valuable information about Prime products prior to major releases or complete revisions of a book. It is a good practice, when inquiring about documentation, to ask about any updates that may be available for a given product.

The PRIMOS HELP command provides information about PRIMOS-level MIDASPLUS commands such as CREATK and KBUILD.

PRIME DOCUMENTATION CONVENTIONS

The following conventions are used in command formats, statement formats, and in examples throughout this document. Examples illustrate the uses of these commands and statements in typical applications. Terminal input may be entered in either uppercase or lowercase.

Convention	Explanation	Example
UPPERCASE	In command formats, words in uppercase indicate the names of commands, options, statements, and keywords. Enter them in either uppercase or lowercase.	MPLUSCLUP
lowercase	In command formats, words in lowercase indicate variables for which you must substitute a suitable value.	key-name
Brackets [ ]	Brackets enclose an optional word or phrase.	[OWNER-IS literal-1]
Braces { }	Braces enclose a vertical list of items. Choose one and only one of these items.	{ SEQUENTIAL RANDOM DYNAMIC }
Ellipsis ...	An ellipsis indicates that the preceding item may be entered more than once on the command line.	[,filename2, ...]
Default indicator ●	In a list of options, a bullet indicates the default choice, if one exists. If you do not select an option, the system chooses the default option.	ON OFF ●
Underscore in examples	In examples, user input is underscored but system prompts and output are not.	OK, creatk This is the output of MY_PROG.CPL OK,
(CR)	This symbol indicates a RETURN key. (CR) is used in examples to show that the user presses the Return Key and nothing else in response to a MIDASPLUS utility prompt.	INDEX NO.? (CR)

Angle brackets in messages < >	In messages, characters or words enclosed within angle brackets indicate a variable for which the utility substitutes the appropriate value.	DO YOU WANT THE INDEX <#> KEY DUMPED?

The term word in this manual means a 16-bit entity.

1
Introduction

This chapter defines MIDASPLUS and lists its file access methods and the language groups associated with MIDASPLUS. MIDASPLUS, the Enhanced Multiple Index Data Access System, is a collection of subroutines and interactive utilities that construct, access, and maintain keyed data files. Once you have established the structure of a MIDASPLUS file, you can add data to it online through interactive programs, or through application programs. You can also use MIDASPLUS utilities to add data in existing sequential (non-MIDASPLUS) files to MIDASPLUS files. Use MIDASPLUS when:

- You wish to access a large data file by one or more keys. For example, you may access a customer master file by an account number or by a customer name.
- Several users need to access and update a file online simultaneously.

For details on features that are new at this revision of MIDASPLUS, see ABOUT THIS BOOK, which immediately precedes this chapter; the section NEW FEATURES summarizes each new feature and tells you where this book describes it fully.

The MIDASPLUS 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 MIDASPLUS files. The subroutines are integrated into and are indirectly used by BASIC/VM, COBOL, PL/I and VRPG. Users of FORTRAN, C, Pascal, and PL/I can call these subroutines directly from programs

written in these languages. (PL/I can call the subroutines either directly or indirectly.) See Chapter 5, THE FORTRAN INTERFACE, for information about the subroutines.

ACCESSING MIDASPLUS

You can access MIDASPLUS files through the following Prime languages:

F77 or FTN *
COBOL 74
BASIC/VM
PL/I
VRPG
C *
Pascal *

* You can access these languages using a call interface only. See Appendix G, THE CALL INTERFACE WITH C, for more information about using C with a call interface and Appendix H, THE CALL INTERFACE WITH PASCAL, for more information about using Pascal with a call interface.

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

MIDASPLUS 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, such as a last name or identification number, which describes or pertains to an individual event, person, company, and so forth. Each record in a file has the same field layout. It is important to remember that the actual contents of each field (called the field value) will usually differ from record to record. As a result, at least one field in each record must have a unique value that distinguishes it from all other file records.

Some file records contain fields that identify the record and fields that describe the record. The fields that identify a record are called key fields or keys. These fields are distinguished from other fields in the record that contain descriptive data, or detail information. Files with key fields are called keyed files.

Some MIDASPLUS dialogs ask if you wish to use secondary data. Secondary data is data that is stored in a secondary index subfile. Since you cannot access secondary data in the same way as ordinary data from the data subfile, the use of secondary data is not recommended.

Keys

Each MIDASPLUS file must contain a primary key and may contain secondary keys. A primary key must have a unique value for each record in the data file. A secondary key is not required to have a unique value in every record. There may be as many as 17 secondary keys for each MIDASPLUS file record. See Figure 1-1 for a diagram of a MIDASPLUS file.

The data file, also called a data subfile, consists of records that you can reference 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. If secondary keys are used, they can also reference the entries in the data subfile.

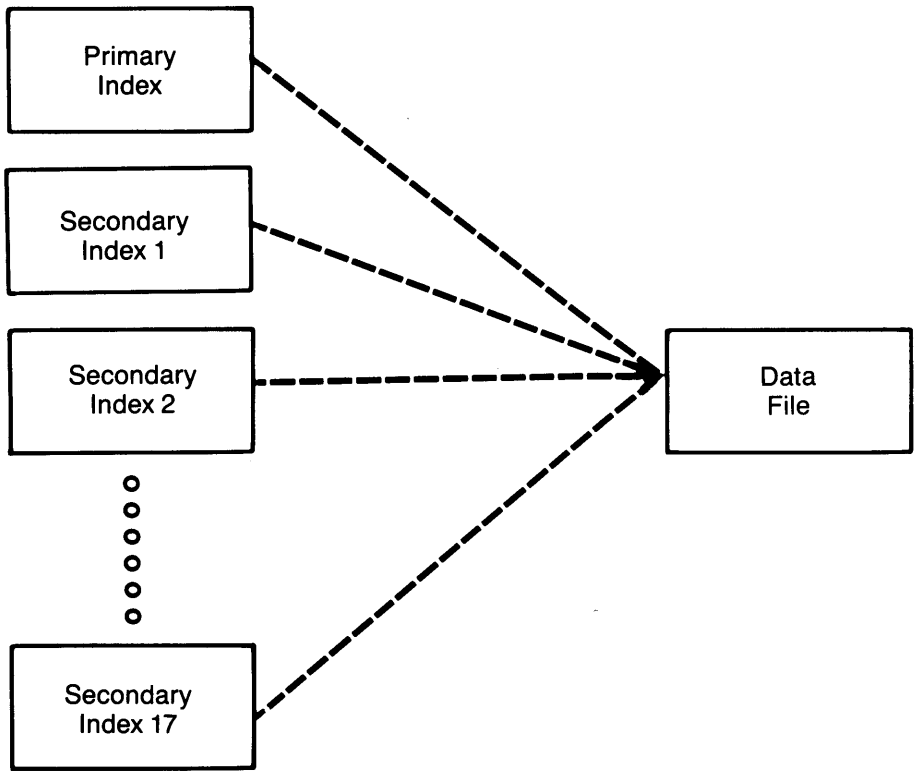


Figure 1-1
Sample MIDASPLUS File

The MIDASPLUS Template

Although the exact number of files and subfiles varies, a MIDASPLUS file always consists of index subfiles (one for each key in the file) and the data subfile, which contains the information to be accessed. Together, these two parts are called a template. A template is essentially an initialized (unpopulated or empty) MIDASPLUS file.

A template's primary function is to accurately define the structure and properties of a MIDASPLUS data file. MIDASPLUS utilities and access routines require a template in order to access the information in a data file. The template includes a description of the data file's key types and their lengths, as well as the data file record length. The record length can be fixed or variable. Variable-length records use only the space needed to contain the data, thereby saving you space.

The MIDASPLUS utility CREATK sets up a MIDASPLUS file template using interactive dialog. See Chapter 2, CREATING A MIDASPLUS FILE, for a description of CREATK.

MIDASPLUS FILE ACCESS METHODS

You can use either the keyed-index access method or the direct access method to retrieve information from a MIDASPLUS file. You can set up a MIDASPLUS file template for keyed-index access only, or you can set it up to use both methods. The MIDASPLUS file structure and access is key-oriented, which makes it easier for you to maintain, update, and retrieve information stored in both large and small files.

Keyed-index Access

Keyed-index access involves giving MIDASPLUS a primary or secondary key value and waiting for MIDASPLUS to return the appropriate record. MIDASPLUS does keyed-index file searches by looking through a list of index subfile entries for a match on the user-supplied key value. When 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. You can do sequential searches by performing a get next-record operation, which tells MIDASPLUS to return the next record entry in the data subfile. You can do partial searches by using the 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, you must give MIDASPLUS a record number. Although you 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. In FORTRAN, you can access direct access files either by record number, primary key, or secondary key.

For direct access files with primary and secondary keys in addition to record numbers, you can use the keyed-index access method to retrieve information by key value. This means that the keyed-index access method can be used on files of either type of template, while direct access only works on templates set up for direct access. You may use direct access with the COBOL, FORTRAN, and VRPG MIDASPLUS interfaces.

EXECUTE-ONLY MIDASPLUS

Prime offers a reduced price version of MIDASPLUS with reduced functionality, which you can use to execute prepackaged MIDASPLUS programs. With execute-only MIDASPLUS, you cannot create new files or new application programs, but you can use CREATK to examine and maintain existing files. With this version of MIDASPLUS, calls to the file creation routine KX$CRE have no effect, because KX$CRE is absent, and no MIDASPLUS libraries are supplied for building an application.

Because execute-only MIDASPLUS offers limited functionality, only part of this book applies this version of the product. The applicable chapters are Chapters 1, 3, 10 through 16, and 18.

LANGUAGE GROUPS

Although you may use any of Prime's languages to access a MIDASPLUS file, the languages with built-in interfaces have some limitations when you are using MIDASPLUS. 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 pertaining to each language interface. Other restrictions pertaining to file access and maintenance are addressed separately in each of the language interface chapters.

FORTRAN

Because FORTRAN is the principal MIDASPLUS interface, and is the basis of all of the other language interfaces, FORTRAN users can take advantage of the full range of MIDASPLUS features. You can create up to 17 secondary keys (and index subfiles) per file. Although keys do not have to be part of the data record, including them in the data record makes it easier to monitor file integrity. To include them, define each key as an actual field in the record. See Chapter 5, THE FORTRAN INTERFACE, for more information about FORTRAN.

COBOL

The COBOL interface to MIDASPLUS uses the Prime CBL compiler and is based on the standard COBOL I/O statements for INDEXED and RELATIVE files. A keyed-index MIDASPLUS file, called an INDEXED SEQUENTIAL file in COBOL, can have one primary key and up to 17 secondary keys. Direct access MIDASPLUS files are also available for COBOL use; these files are called RELATIVE files in COBOL.

BASIC/VM

MIDASPLUS files built for access by BASIC/VM programs can have one primary and up to 17 secondary keys. Although keys are not required to be part of the data record, it is recommended that you include both primary and secondary keys in the data record for convenience. BASIC/VM does not support the direct access feature of MIDASPLUS. See Chapter 7, THE BASIC/VM INTERFACE, for additional information about BASIC/VM.

PL/I

The PL/I MIDASPLUS interface supports only ASCII primary keys, with a maximum length of 32 characters. PL/I does not support secondary keys or direct access. It is not necessary to use CREATK to set up a MIDASPLUS file template, as PL/I has its own tools for doing so. You can access files created with CREATK, however, through PL/I. See Chapter 8, THE PL/I INTERFACE, for more information about PL/I.

VRPG

The VRPG interface to MIDASPLUS supports up to 17 secondary keys for keyed-index files, but does not support the use of secondary data. The keys may be of type ASCII or bit string. VRPG supports access to both keyed-index and direct access MIDASPLUS files, but can only delete keyed-index MIDASPLUS records. See Chapter 9, THE VRPG INTERFACE, for more information about VRPG.

2

Creating a MIDASPLUS File

This chapter tells you how to create keyed-index access and direct access MIDASPLUS files, explains the CREATK dialogs, provides examples of the CREATK dialogs, and summarizes other features of CREATK that you can use on already existing MIDASPLUS files. CREATK is an interactive program that uses your specifications in the form of parameters to set up a template that describes a MIDASPLUS file and allocates space for it. These parameters include the following:

- MIDASPLUS file type (keyed-index or direct access)
- Primary key type and size
- Secondary key types and sizes - optional. Use secondary keys when you want more than one search key for the file
- Data record size

DIALOGS

CREATK has two dialogs - the minimum options dialog and the full options dialog. The minimum options dialog supplies default parameters for the template. With the extended options dialog, you provide all of the parameters to build a template. Both of these options allow you to build a keyed-index or direct access MIDASPLUS file.

While the minimum options dialog only asks a few questions about the file and its keys, the extended options dialog asks for more details, such as segment length and index block size.

The default parameters give the best performance. Use the extended options dialog only if you want to change the index block size. (See Chapter 14, ADDITIONAL CREATK FUNCTIONS, for information about the extended options dialog.)

Dialog Guidelines

As with other MIDASPLUS utilities, input to CREATK can be in either lowercase or uppercase. If you make an input error, CREATK displays a message telling you what the problem is. This message repeats until you enter acceptable input. When CREATK asks a "YES/NO" type of question, it accepts the following responses:

YES
NO
AYE
NAY
OK

CREATK also accepts the first letter of the above responses (Y, N, A, or O). To end CREATK's dialog, press the RETURN key after the INDEX NO? prompt appears.

Note

User responses are underlined in this manual to distinguish your responses from system output. Never underline your input. The (CR) symbol shown in the examples indicates the RETURN key. Enter CREATK to begin the CREATK dialog. See the keyed-access and direct access dialogs and examples later in this chapter.

Key Types

Table 2-1 lists the data types for MIDASPLUS keys. 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. In the MIDASPLUS dialog, the term word refers to 16 bits.

Table 2-1
MIDASPLUS File Key Types

Key Code	Key Type	Length Specifications
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. 16 words (256 bits)
D	Double Precision Floating Point (REAL*4)	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 (REAL*2)	Hardware-defined: 2 words

FILE READ/WRITE LOCKS

CREATK automatically sets the file Read/Write lock on each MIDASPLUS file it creates to n readers and n writers. This setting is equivalent to the PRIMOS RWLOCK setting of 3. With a lock setting of 3, multiple users may have the file open for reading, writing, or updating. The Read/Write lock settings are part of MIDASPLUS concurrent process handling.

CREATK displays the message:

SETTING FILE LOCKS TO N READERS AND N WRITERS

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

SAMPLE FILE

Figure 2-1 shows the layout of a sample MIDASPLUS file called BANK, which is used for examples throughout this book. This file is designed to provide information about a bank's customers and their accounts. The file consists of the following three major fields:

customer identification number
customer name
account number

The BANK file is a keyed-index access file created with one primary key and two secondary keys. The primary key is an ASCII key, nine characters in length, describing a customer's identification number. The first secondary key is an ASCII key of 25 characters describing the customer's name. The second secondary key, a ten-character ASCII string, describes the customer's account number. The non-keyed fields include street address, city, state, and zip code.

Field 1	Field 2	Field 3	Field 4
CUSTOMER ID	CUSTOMER NAME	ACCOUNT NUMBER	ADDRESS
• Primary key	• Secondary key 01	• Secondary key 02	• Not a keyed field
• 9 characters – duplicates not allowed	• 25 characters – duplicates allowed	• 10 characters – duplicates not allowed	• 42 characters
• ASCII key	• ASCII key	• ASCII key	
Record length = 86 characters			

Layout of a BANK File
Figure 2-1

VARIABLE-LENGTH RECORDS AND SPACE USAGE

If you create a file with records of fixed length, but with each record containing a different amount of data, some disk space is wasted. For instance, if you indicate that each record will be 36 characters long, but record X contains only 24 characters, a third of the record's disk space is unused. For this situation, MIDASPLUS allows you to create a file of records that vary in length. Each one of these variable-length records (VLRs) uses only the disk space it needs to contain the data.

To create and load a file of variable-length records, use CREATK and load the file as follows:

1) Begin the CREATK dialog, indicating that you are creating a keyed-indexed access file, the only type of MIDASPLUS file that may contain variable-length records.

2) When CREATK prompts DATA SIZE IN WORDS, respond in one of the following ways:

   - Press RETURN or enter a 0. Either action prohibits CREATK from setting a minimum or a maximum limit on the record size.

   - Enter a 0 followed by two values. The first value sets the minimum record size; the second value sets the maximum record size. The minimum must be at least 1; the maximum can be up to 32767. Setting these size limits accommodates applications that check size limits when opening a file or before adding a record.

3) Load the file according to the instructions in BUILDING A VARIABLE-LENGTH RECORD MIDASPLUS FILE in Chapter 3.

Three CREATK commands, INITIALIZE, GET, and PRINT, help you administer variable-length record files. If you created the file without setting size limits and the file is still empty, use the INITIALIZE command to set size limits. You can also use INITIALIZE to change these limits, before or after you load the file. If you loaded the file before setting size limits, use the GET command, which sets the limits to the size of the smallest and largest records in the file. To find out the current size limits, issue the PRINT command and look for a line with the following format:

VLR MIN SIZE: <number> VLR MAX SIZE: <number>

KEYED ACCESS DIALOG (MINIMUM OPTIONS)

This section consists of the prompts and responses for the CREATK keyed-index access dialog with minimum options. Remember that in the MIDASPLUS dialog, the term word refers to 16 bits (2 bytes).

Prompt	Response
MINIMUM OPTIONS?	Enter YES.
FILE NAME?	Enter the pathname of the file to be created.
NEW FILE?	Enter YES to create a new template.
DIRECT ACCESS?	Enter NO to create a keyed-index access file.
DATA SUBFILE QUESTIONS	
PRIMARY KEY TYPE:	Enter one of the key codes listed in Table 2-1 to define the primary key data type (A, B, D, I, L, or S).
PRIMARY KEY SIZE = :	Enter the size of key in words, bytes, or bits. Size must be preceded by W and a space for words or B and a space for bytes or bits. See Table 2-1.
DATA SIZE IN WORDS = :	For fixed-length records, enter the maximum length in words of the data record in the data subfile. If the MIDASPLUS files will be used in COBOL applications, include the total length of all keys in the data size. For variable-length records, either press RETURN, or enter 0, or enter 0 followed by a value for the minimum record size and a value for maximum record size.
SECONDARY INDEX	
INDEX NO.?	For FORTRAN, BASIC/VM, VRPG and COBOL applications, enter a number from 1-17 to indicate which secondary index is being defined. For COBOL applications, define the secondary keys in order, that is, define secondary key 1 before secondary key 2, and so forth.

For use with PL/I applications, press the RETURN key. (PL/I does not support secondary keys.)

Enter 0 or press the RETURN KEY to end the secondary index definition sequence.

DUPLICATE KEYS PERMITTED? — Enter YES or NO. YES allows the same secondary key value to appear more than once in the index.

KEY TYPE: — Enter one of the codes listed in Table 2-1 to indicate data type of the secondary key (A, B, D, I, L, or S).

KEY SIZE = : — Enter the size of key in words, bytes, or bits. Size must be preceded by W and a space for words or B and a space for bytes or bits. (Asked only if A or B type key is specified above.)

SECONDARY DATA SIZE IN WORDS = : — For use with FORTRAN, enter the number of words of secondary data to be stored with this secondary key or press the RETURN key. The use of secondary data is not recommended.

For other languages, enter 0 or press the RETURN key.

Note

The secondary index prompts repeat, enabling you to enter information about each secondary index. To complete the CREATK process, press RETURN at the INDEX NO? prompt.

Keyed Access Example

```
OK, creatk
[CREATK rev 19.4.0]

MINIMUM OPTIONS? yes

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

DATA SUBFILE QUESTIONS

PRIMARY KEY TYPE: a
PRIMARY KEY SIZE = : b 9
DATA SIZE IN WORDS = : 43

SECONDARY INDEX

INDEX NO.? 1

DUPLICATE KEYS PERMITTED? yes

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

INDEX NO.? 2

DUPLICATE KEYS PERMITTED? no

KEY TYPE: a
KEY SIZE = : b 10
SECONDARY DATA SIZE IN WORDS = : (CR)

INDEX NO.? (CR)

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

DIRECT ACCESS DIALOG (MINIMUM OPTIONS)

This section consists of the prompts and responses for using the direct access dialog with minimum options for CREATK. Direct access, which is available in FORTRAN, COBOL, and VRPG, is discussed in Chapters 5, 6, and 9.

Prompt	Response
MINIMUM OPTIONS?	Enter YES.
FILE NAME?	Enter the pathname of the file to be created.
NEW FILE?	Enter YES to create a new template.
DIRECT ACCESS?	Enter YES to create a direct access file. Records are stored in the data subfile in sequential order according to record number.
DATA SUBFILE QUESTIONS	
PRIMARY KEY TYPE:	Enter B or A for files used with COBOL. Enter A for files used with VRPG. Enter one of the codes listed in Table 2-1 for FORTRAN (A, B, D, I, S, or L).
PRIMARY KEY SIZE =	Enter the size of the key in words, bytes, or bits. Size must be preceded by W and a space for words or B and a space for bytes and bits. See Table 2-1.
DATA SIZE IN WORDS = :	Enter the length of the data record in the data subfile. Direct access files must have fixed-length records supplied in 16-bit words.
NUMBER OF ENTRIES TO ALLOCATE?	CREATK preallocates space for direct access files. Enter the maximum number of entries (records) that you expect to use in the data subfile.

SECONDARY INDEX

INDEX NO.?	For use with FORTRAN or VRPG, enter a number from 1-17 if secondary keys are desired. For use with COBOL, press the RETURN KEY.
DUPLICATE KEYS PERMITTED?	Enter YES or NO. YES allows the same secondary key value to appear more than once in the index.
KEY TYPE:	Enter one of the codes listed in Table 2-1, to indicate data type of the secondary key (A, B, D, I, S, or L).
KEY SIZE = :	Enter the size of key in words, bytes, or bits. Size must be preceded by W for words or B for bytes or bits. See Table 2-1.
SECONDARY DATA SIZE IN WORDS = :	Enter 0 or press the RETURN key to specify no secondary data. For use with FORTRAN, enter the number of words to be stored with this secondary key. The use of secondary data is not recommended.

Note

The secondary index prompts repeat, enabling you to enter information about each secondary index file. To complete the CREATK process, press the RETURN key at the INDEX NO? prompt.

Direct Access Example

```
OK, creatk
[CREATK rev 19.4.0]

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 IN WORDS = : 17
NUMBER OF ENTRIES TO ALLOCATE? 15

SECONDARY INDEX

INDEX NO.? (CR)

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

OPTIONAL CREATK FEATURES

CREATK also lets you get information about an existing MIDASPLUS file, its key types and sizes, its index subfile structure, segment length, block size, and so forth. You can also change the length of the data file record and get estimates on how much room is needed for a projected number of files.

If you enter CREATK and type NO to the NEW FILE? prompt, the next prompt will be FUNCTION?. To obtain a list of these CREATK functions, type H (HELP) after the FUNCTION? prompt appears on your screen. These functions can only be used on existing files. For example:

```
OK, creatk
[CREATK rev 19.4.0]

MINIMUM OPTIONS? yes

FILE NAME? bank
NEW FILE? no
```

```
FUNCTION? help

A[DD]          =  ADD AN INDEX
C[OUNT]        =  COUNT ACTUAL INDEX ENTRIES
D[ATA]         =  CHANGE DATA RECORD SIZE
E[XTEND]       =  CHANGE SEGMENT & SEGMENT DIRECTORY LENGTH
F[ILE]         =  OPEN A NEW FILE
G[ET]          =  GET AND SET THE ACTUAL MIN/MAX RECORD SIZE OF THE
                  VARIABLE LENGTH RECORD (VLR) FILE
H[ELP]         =  PRINT THIS SUMMARY
I[NITIALIZE]   =  SET THE MIN/MAX RECORD SIZE FOR THE VARIABLE
                  LENGTH RECORD (VLR) FILE
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]      =  MIDASPLUS DEFAULTS FOR THIS FILE
```

Explanation of CREATK Options

The CREATK options are summarized below and described in more detail in Chapter 14, ADDITIONAL CREATK FUNCTIONS.

OPTION	USE
ADD	Allows you to add a secondary index subfile and a key to an existing MIDASPLUS template. You cannot have more than 17 secondary index subfiles.
COUNT	Counts the number of entries currently in the file.
DATA	Changes the data record length and the number of records allocated for that file if the file is a direct access file. DATA does not display the current record length. (Use the PRINT option to get the current record length.)
EXTEND	Lets you change the number of segments per segment directory and words per segment subfile. With EXTEND, you can extend the segment subfile and segment directory lengths; this allows you to make bigger index and data subfiles.
FILE	Lets you create a new file template without returning to PRIMOS and reentering CREATK, or lets you work on another old file. Returns you to PRIMOS after file definition is complete.

GET — Displays the sizes of the largest and smallest records in a variable-length record file. Also sets or changes record size limits in the three situations: a) if minimum and maximum record sizes are not set, GET sets them to the sizes of the largest and smallest records in the file; b) if size limits are set, but the smallest record is smaller than the minimum size, GET changes the minimum to that record's size. c) if size limits are set, but the largest record is larger than the maximum size, GET changes the maximum to that record's size.

HELP — Displays the list of functions.

INITIALIZE — Lets you set the minimum and maximum record sizes for a variable-length record file before you load it. (If you already loaded the file, but size limits are not set, use the GET command to set them.) Before or after you load the file, INITIALIZE lets you decrease the minimum or increase the maximum record sizes.

MODIFY — Allows you to change support of duplicate keys and change secondary data size. You can change the index block length if you use the extended options feature of CREATK.

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 index level, describes the entry size, block size, control words, maximum number of entries per block, and the number of blocks in that level. PRINT displays data subfile information, as of the last MPACK, including the file access type (keyed or direct), the number of indexes, the entry size, and the key size. For a variable-length record file, PRINT also displays the minimum and maximum record sizes, if these size limits are set.

QUIT — Exits the CREATK dialog and returns you to PRIMOS. (Pressing the carriage return does the same thing.)

SIZE — Estimates the number of segments and disk records required for a hypothetical number of entries. Estimates size for each index, for the data records, or the total file.

USAGE — Provides information on the total number of entries in the file, the number of entries indexed, the number of entries deleted, and the number of entries inserted since the last MPACK. Also displays the version of MIDASPLUS which last modified the file.

VERSION Displays the revision stamp of the version of MIDASPLUS under which this file was created and the default parameters for the file (for example, DAM file length or segment directory length).

Note

If CREATK cannot get exclusive use of the ADD, DATA, EXTEND, or MODIFY options, the following message appears:

```
THIS FILE IS IN USE.  AVAILABLE OPTIONS ARE:
  F[ILE]      =  OPEN A NEW FILE
  H[ELP]      =  PRINT THIS SUMMARY
  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]   =  MIDASPLUS DEFAULTS FOR THIS FILE
```

3

Building a MIDASPLUS File

This chapter discusses input files and the location of keys with KBUILD, explains how to use KBUILD to build keyed or direct access MIDASPLUS files, and presents sample KBUILD dialog sessions.

Using the KBUILD utility is one method of adding records to a MIDASPLUS file. (Other methods of adding records are by using application programs, offline routines, and PRIME/POWER+.) KBUILD is the recommended method when you wish to add a large number of records at one time. The functions of KBUILD include:

- Adding data to a new (that is, "empty") MIDASPLUS file template. This includes adding entries to the needed index subfiles from sorted or unsorted input data.
- Adding new data and index entries to an existing MIDASPLUS file that already contains data entries.
- Building keyed-index MIDASPLUS files containing either fixed-length or variable-length records.
- Building direct access MIDASPLUS files.
- Adding entries from an external data file to a new secondary index subfile that was defined for a previously populated MIDASPLUS file.
- Converting a field from existing MIDASPLUS data subfile records to a secondary key field. (These entries are added to a new or already existing secondary index subfile.)

Use KBUILD to add a large number of records to a MIDASPLUS file. KBUILD adds the primary index entry, the data subfile entry, and the supplied secondary index entries for each record. If you are adding many entries to a file, keep a copy of the input file(s) in case the MIDASPLUS file is damaged or the system crashes. An easy way to regenerate a file is to set up a command file that first invokes CREATK to set up the template; then invoke KBUILD to populate the file.

Note

KBUILD zeroes out secondary data and cannot handle concatenated keys.

INPUT FILES

While KBUILD only supports input files with fixed-length records, you can use specially formatted fixed-length input files to build MIDASPLUS files with variable-length records. See Building a Variable-length MIDASPLUS File later in this chapter for more information.

KBUILD can handle input files created by various methods. Table 3-1 lists KBUILD supported files and their file type codes. Before beginning the KBUILD dialog, make sure that you know the type of file that you are using. KBUILD needs this information to process the input correctly.

Table 3-1
KBUILD-Supported Input File Types

File Type Code	Description
BINARY	A binary file created by PRWF$$, which is usually called from a FORTRAN program. Such a file has no newline characters (.NL.).
COBOL	An uncompressed file of fixed-length records containing ASCII or binary data and delimited by a newline character.
FTNBIN	A FORTRAN WRITE statement creates this binary file via the routine O$BD07. FTNBIN is used in FORTRAN as binary output. The first word of each record in this type of file indicates the record's length. It contains no newline characters. A CBL WRITE statement for variable-length records also produces this filetype.
RPG	The file to which O$AD08 routine writes the data. It is an uncompressed file with fixed-length records and newline character delimiters. Each record must contain the primary key but it does not have to be the first field in the record.
TEXT	Any file that is created in, or used by, the editor. A newline character ends the records. Text files are the most common type. A CBL WRITE statement for COMPRESSED records also produces this filetype.

Input File Rules

Input files always have fixed-length records. Input files can contain:

- Primary key values and data values (secondary key values are optional)
- Secondary key values only (include the primary key value with which its record is associated)

Data is defined as the information that is to be written to the data subfile. If you want to use KBUILD to convert existing files to MIDASPLUS files, KBUILD requires the following from input files and their record structure:

- All key elements should be at the beginning of the record.
- No unnecessary data can be located before the fields that you want KBUILD to process.

Specify the starting character position of each key field in the input record. The first character position in the record is character position 1. Begin key fields on byte boundaries. If the key fields are not physically part of the data subfile entry, other key fields may appear after the data.

Tell KBUILD the input record length. The length is the number of words in an input record excluding anything that the file inserts (for example, leading word count in FTNBIN files). Input files are not required to have the same record length as that of output MIDASPLUS files.

Location of Keys

If you do not want the keys to be in the data subfile records, put the keys after the data that you want included in the data subfile entries. KBUILD can truncate the entries when KBUILD writes them to the data subfile. Only the initial portion of the input record (without keys) is written to the MIDASPLUS data subfile.

Record Compatibility

Make sure that all records of the input file have identical record layouts. For example, if the primary key begins in character position 1 of the first record, the primary key should begin in the same position for all of the other records in the file. When writing a record from the input file to the output (MIDASPLUS) file, KBUILD always begins with character position 1 of the input record. The length of the entry written to the output file depends on whether the MIDASPLUS file has fixed-length or variable-length records.

Multiple Input Files

KBUILD lets you process more than one input file during a single run; therefore, you can add information from up to 99 data files to a single MIDASPLUS file, but you must assign special names to the files. If there is more than one input file, begin the filenames with the same

letters and end in a two digit number, representing the sequence in which the files should be processed. For example:

```
BRANCH01
BRANCH02
BRANCH03
```

Make sure all of the files exist in the same directory and have exactly the same format and file type.

Note

While KBUILD can process multiple input files, it can create only one output MIDASPLUS file at a time.

Sorted Input Files

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

The data subfile entries are always stored in the order in which they are read from the input file. If you frequently need to access the data sequentially by a key, sort the input file by that key before using KBUILD. This step will improve the performance of future sequential reads.

You can only add pre-sorted index entries to a primary or secondary index that does not contain entries. If you try to build a non-empty secondary index from a sorted input file, KBUILD informs you of your error. When the index appears to be empty but has keys pointing to deleted records, you must run MPACK on the index to clean it out before you can add sorted input entries to it. If you do not care about the entries that are in the index, you can delete them all at once, by running KIDDEL to clean out the secondary index subfile before trying to add sorted entries to it. (See Chapters 15 and 11, respectively for details on MPACK and KIDDEL.)

Sort Requirements: A file is considered sorted only if a primary or secondary key field is a sort key. In addition, the following rules apply if there are several input files:

- Sort all of the input files on the same field.
- Make sure that all of the sorted key values in the first file are less than the sorted key values of the second file. The same rule holds for the other files.
- If a MIDASPLUS file index to which the entries are being added already contains entries, do not declare the input files sorted by that key even if they are sorted. If the input file is called "sorted", KBUILD does not process an input file for building a MIDASPLUS index.

BUILDING A VARIABLE-LENGTH FILE

KBUILD can take fixed-length records from input files and add them as variable-length records to MIDASPLUS files. Before you use KBUILD in this way, you must perform the two steps below:

1) Determine the number of words in each input record. For instance, if the file contains character data, you would do the following:

   A. For each record, count the number of characters (including blank spaces) that the primary key, secondary keys, and nonkeyed data occupy.

   B. If the number of characters in the record is an odd number, increase the number of characters by 1.

   C. Divide the number of characters by 2 to determine the character size in words of the record.

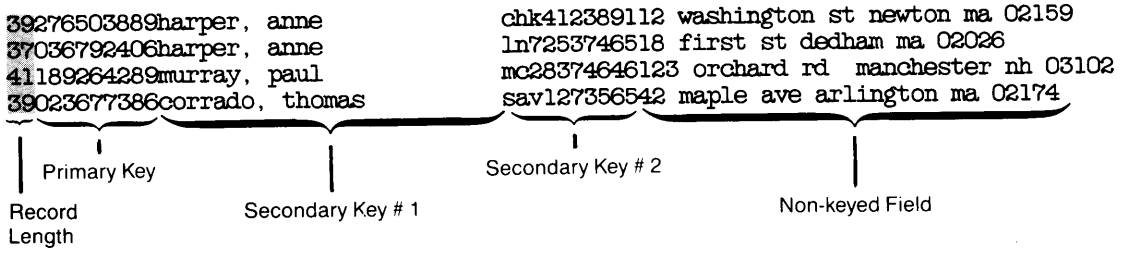
      For example:

      2 First St Dedham MA 02026 = 26 chars./2 = 13 words

      18 First St Dedham MA 02026 = (27 chars. + 1)/2 = 14 words

2) For each record, edit the input file to add the length you determined. Place this number in the same word position in each record, as shown in the following example.

The following example shows an input file that contains the record length in positions one and two, which are highlighted.



This input file, created for the sample banking application, consists of a customer identification number (primary key), name, account number, and address. This file was created using the PRIMOS screen editor (EMACS) and is in ASCII form (TEXT).

After KBUILD determines that your output file has variable-length records, the KBUILD dialog asks whether this number is a bit (B) string or an ASCII (A) string.

Note

> Rather than using KBUILD, you can also build variable-length MIDASPLUS files with PRIBLD, SECBLD, BILD$R, and the standard interface. These routines also support direct access files.

If KBUILD or a build routine adds a variable-length record that is outside a record size limit, MIDASPLUS automatically resets that limit to the size of the record.

ADDING SECONDARY INDEX ENTRIES ONLY

Besides adding all of the primary and secondary key entries with the data subfile entries, KBUILD can also populate a secondary index subfile when

- You decide to make one of the fields in a MIDASPLUS data record a secondary key. (You would do this if you need more keys.)
- You did not supply secondary key values for all of the data entries that you originally added. As a result, there is no one-to-one correspondence between secondary index entries and data subfile entries.

When one of the above statements is true, either make one of the fields in the data record a secondary key or take the secondary index entries from an external input file. (The primary key must be present in the input record.)

Note

You must first use CREATK to define a new index subfile before you can use that subfile with KBUILD. KBUILD alone cannot define a new index subfile.

ERROR REPORTING

KBUILD reports all fatal and non-fatal errors that it finds during processing. KBUILD prints the type and number of the error, as well as the record number that was being processed when the error occurred. The log/error file records fatal errors just before KBUILD aborts.

All errors are displayed at your terminal. KBUILD also displays the name of the input file that it is processing and tells you what part of the MIDASPLUS file that it is currently in. If you want to record this data, enter a new file name when KBUILD prompts you for a log/error file name; otherwise, press the RETURN key.

Milestone Reporting

KBUILD reports milestone statistics at your terminal. If you want to record this data, enter a new file name when KBUILD prompts you for a log/error file name; otherwise, press the RETURN key. Sample milestone reports are included with the KBUILD examples. A milestone report consists of:

- The record number for which the milestone is generated
- The current date and time
- The CPU and disk time used since KBUILD began to process the file
- The total disk and CPU time elapsed since the start of KBUILD's run
- The amount of time elapsed since the last milestone report was created

If the input file is unsorted, KBUILD tells you when the file begins and ends a sort pass through each set of index entries. For large files, set the milestone count according to how concerned you are with resource usage.

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

THE KBUILD DIALOG

Type KBUILD to invoke the KBUILD utility. The KBUILD dialog and an explanation are included in this section. The dialog is numbered in this book for explanatory purposes only.

PROMPT	RESPONSE
1. SECONDARIES ONLY?	YES or NO YES - Builds/adds entries to one or more of the secondary index subfiles. Subfiles need not be empty. The dialog continues at step 2. NO - Adds data entries to data subfiles in primary key order. Also adds entries to the primary index subfile and any secondary index subfiles as indicated. The dialog continues at step 4.
2. USE MIDASPLUS DATA?	YES or NO YES - Uses existing data entries as a source of values for a secondary index subfile when you have existing records in the data subfile and you want to make fields from these records into secondary keys. The dialog continues at step 3. NO - All subfile and data subfile entries are taken from an input file (not a MIDASPLUS file) which must contain primary key values. The dialog continues at step 4.
3. ENTER MIDASPLUS FILENAME:	The MIDASPLUS pathname from which MIDASPLUS should get the secondary key entries. The dialog continues at step 14.
4. ENTER INPUT FILENAME:	The name of the input file that KBUILD will process. If you are using multiple files, enter the name of the file with the lowest sequence number.
5. ENTER INPUT RECORD LENGTH (WORDS):	The size in 16-bit words of the input file record.

6. INPUT FILE TYPE:	The appropriate KBUILD code (BINARY, COBOL, FTNBIN, RPG, TEXT).
7. ENTER NUMBER OF FILES:	1 for single files. The total number of files for multiple input files.
8. ENTER OUTPUT FILENAME:	The pathname of the MIDASPLUS file to which you will add data in the input file. If the MIDASPLUS file has fixed-length records, the dialog continues at step 13. Otherwise, the dialog continues with the next step.
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*) STRING? (ENTER A OR B):	A, if the Editor, COBOL, or VRPG created the input file and the output record length is in ASCII form. The dialog continues with step 10. B if the file is BINARY or FTNBIN. The dialog continues with step 12.
10. ENTER STARTING CHARACTER POSITION IN INPUT RECORD:	The character position where the output record length specification begins.
11. ENTER ENDING CHARACTER POSITION IN INPUT RECORD:	The character position that marks the end of the output record length specification. Dialog continues with step 13.
12. ENTER STARTING WORD NUMBER IN INPUT RECORD:	The word number in the input record record that specifies the output record length for Binary (INT*2) representations.
13. ENTER STARTING CHARACTER POSITION, PRIMARY KEY:	The starting position of the input record field containing a primary key value.
14. SECONDARY KEY NUMBER:	The number of the secondary index entry for which you will take an entry from the input file record.
15. ENTER STARTING CHARACTER POSITION:	The character position in the input record where the secondary key field begins.

Note

Prompts 14 and 15 are repeated until you press the RETURN key.

16. IS THE FILE SORTED?

NO if the file is unsorted. The dialog continues with step 19.

YES if the file is sorted.

17. IS THE PRIMARY KEY SORTED?

YES if the input file is sorted by the primary key field.

NO if the input file is not sorted by the primary key field.

18. ENTER INDEX NUMBER OF SECONDARY SORT KEY:

If the file was sorted on a field corresponding to a secondary key, enter that key number. This prompt is repeated until you press the RETURN key.

19. ENTER LOG/ERROR FILE NAME:

The name of the file in which you will record errors and KBUILD milestone statistics.

Make sure that this file name is different from all existing files.

Press the RETURN key if you do not want to record the statistics. The statistics still appear on your terminal.

20. ENTER MILESTONE COUNT:

The frequency (number of records) at which you want the records displayed and recorded in a log/error file.

If you enter 0, milestones are printed for the first and last records of the input file only.

KBUILD EXAMPLES

This section includes KBUILD examples showing the following features:

- Building a fixed-length record MIDASPLUS file from unsorted input
- Building a fixed-length MIDASPLUS file from sorted input

- Building a variable-length record MIDASPLUS file from unsorted input
- Building a fixed-length direct access MIDASPLUS file from unsorted input

User input in the examples is underlined to distinguish it from system prompts and messages.

Example 1: Using Unsorted Input

This example uses an unsorted input file to build entries for the BANK file, which has fixed-length records of 22 words. Input files are not required to have the same record length as that of the output MIDASPLUS files. If the input files are too long when added, they are truncated. If the records are shorter than the specifications in the MIDASPLUS template, the records are blank-padded to the correct length. The input file contains the following records:

```
276503889harper, anne          chk412389112 washington stnewton        ma02159
036792406harper, anne          ln7253746518 first st     dedham         ma02026
189264289murray, paul          mc28374646123 orchard rd  manchester     nh03102
023677386corrado, thomas       sav127356542 maple ave    arlington      ma02174
```

Primary Key | Secondary Key # 1 | Secondary Key # 2 | Non-keyed Field

The file was built with the following dialog:

```
OK, kbuild
[KBUILD rev 19.4.0]

SECONDARIES ONLY? no
ENTER INPUT FILENAME: names
ENTER INPUT RECORD LENGTH (WORDS): 43
INPUT FILE TYPE: text
ENTER NUMBER OF INPUT FILES: 1
ENTER OUTPUT FILENAME: bank
ENTER STARTING CHARACTER POSITION, PRIMARY KEY: 1
SECONDARY KEY NUMBER: 1
ENTER STARTING CHARACTER POSITION: 10
SECONDARY KEY NUMBER: 2
ENTER STARTING CHARACTER POSITION: 35
SECONDARY KEY NUMBER:  (CR)
IS FILE SORTED? no
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  01-11-85  13:52:31        0.000      0.000      0.000    0.000
     1  01-11-85  13:52:31        0.002      0.000      0.002    0.002
     2  01-11-85  13:52:31        0.003      0.000      0.003    0.001
     3  01-11-85  13:52:31        0.004      0.000      0.004    0.001
     4  01-11-85  13:52:32        0.004      0.000      0.004    0.001
FIRST BUILD/DEFER PASS COMPLETE.
     4  01-11-85  13:52:32        0.005      0.000      0.005    0.001

SORTING INDEX 0
 COUNT      DATE      TIME      CPU MIN   DISK MIN   TOTAL TM     DIFF
     0  01-11-85  13:52:32        0.000      0.000      0.000    0.000
SORT COMPLETE
     4  01-11-85  13:52:32        0.006      0.000      0.006    0.006
```

```
BUILDING INDEX 0
 COUNT      DATE      TIME      CPU MIN    DISK MIN    TOTAL TM      DIFF
     0  01-11-85  13:52:32        0.000       0.000       0.000       0.000
     1  01-11-85  13:52:32        0.001       0.000       0.001       0.001
     2  01-11-85  13:52:32        0.002       0.000       0.002       0.001
     3  01-11-85  13:52:32        0.002       0.000       0.002       0.001
     4  01-11-85  13:52:32        0.003       0.000       0.003       0.001
INDEX 0 BUILT
     4  01-11-85  13:52:32        0.004       0.000       0.004       0.001

SORTING INDEX  1
 COUNT      DATE      TIME      CPU MIN    DISK MIN    TOTAL TM      DIFF
     0  01-11-85  13:52:32        0.000       0.000       0.000       0.000
SORT COMPLETE
     4  01-11-85  13:52:33        0.005       0.000       0.005       0.005

BUILDING INDEX   1
 COUNT      DATE      TIME      CPU MIN    DISK MIN    TOTAL TM      DIFF
     0  01-11-85  13:52:33        0.000       0.000       0.000       0.000
     1  01-11-85  13:52:33        0.002       0.000       0.002       0.002
     2  01-11-85  13:52:33        0.002       0.000       0.003       0.001
     3  01-11-85  13:52:33        0.003       0.000       0.003       0.001
     4  01-11-85  13:52:33        0.003       0.000       0.004       0.001
INDEX  1 BUILT
     4  01-11-85  13:52:33        0.004       0.000       0.005       0.001

SORTING INDEX  2
 COUNT      DATE      TIME      CPU MIN    DISK MIN    TOTAL TM      DIFF
     0  01-11-85  13:52:33        0.000       0.000       0.000       0.000
SORT COMPLETE
     4  01-11-85  13:52:34        0.005       0.000       0.005       0.005

BUILDING INDEX   2
 COUNT      DATE      TIME      CPU MIN    DISK MIN    TOTAL TM      DIFF
     0  01-11-85  13:52:34        0.000       0.000       0.000       0.000
     1  01-11-85  13:52:34        0.002       0.002       0.004       0.004
     2  01-11-85  13:52:34        0.003       0.002       0.005       0.001
     3  01-11-85  13:52:34        0.003       0.002       0.005       0.001
     4  01-11-85  13:52:34        0.004       0.002       0.006       0.001
INDEX  2 BUILT
     4  01-11-85  13:52:34        0.005       0.002       0.007       0.001

KBUILD COMPLETE.

OK,
```

Example 2: Using Sorted Input

The following BANK.SORT file is built from sorted input records (sorted according to the primary key). Since KBUILD is not required to sort the file, the build is faster. The input file contains the following records:

```
023677386corrado, thomas      sav127356542 maple ave      arlington      ma02174
036792406harper, anne         ln7253746518 first st       dedham         ma02026
189264289murray, paul         mc28374646123 orchard rd    manchester     nh03102
276503889harper, anne         chk412389112 washington st  newton         ma02159
```

The following dialog was used to build the file:

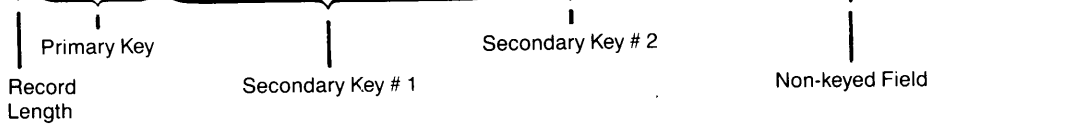
```
OK, kbuild
[KBUILD rev 19.4.0]

SECONDARIES ONLY? no
ENTER INPUT FILENAME: names.sort
ENTER INPUT RECORD LENGTH (WORDS): 43
INPUT FILE TYPE: text
ENTER NUMBER OF INPUT FILES: 1
ENTER OUTPUT FILENAME: bank.sort
ENTER STARTING CHARACTER POSITION, PRIMARY KEY: 1
SECONDARY KEY NUMBER: 1
ENTER STARTING CHARACTER POSITION: 10
SECONDARY KEY NUMBER: 2
ENTER STARTING CHARACTER POSITION: 35
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, 0
DEFERRING: 1, 2
```

```
PROCESSING FROM:  names.sort
 COUNT      DATE      TIME      CPU MIN    DISK MIN    TOTAL TM      DIFF
     0  01-10-85  14:36:19        0.000       0.000       0.000       0.000
     1  01-10-85  14:36:19        0.002       0.000       0.002       0.002
     2  01-10-85  14:36:19        0.002       0.000       0.002       0.001
     3  01-10-85  14:36:19        0.003       0.000       0.003       0.001
     4  01-10-85  14:36:19        0.004       0.000       0.004       0.001
FIRST BUILD/DEFER PASS COMPLETE.
     4  01-10-85  14:36:19        0.005       0.000       0.005       0.001

SORTING INDEX  1
 COUNT      DATE      TIME      CPU MIN    DISK MIN    TOTAL TM      DIFF
     0  01-10-85  14:36:19        0.000       0.000       0.000       0.000
SORT COMPLETE
     4  01-10-85  14:36:20        0.005       0.000       0.005       0.005

BUILDING INDEX   1
 COUNT      DATE      TIME      CPU MIN    DISK MIN    TOTAL TM      DIFF
     0  01-10-85  14:36:20        0.000       0.000       0.000       0.000
     1  01-10-85  14:36:20        0.002       0.000       0.002       0.002
     2  01-10-85  14:36:20        0.002       0.000       0.002       0.001
     3  01-10-85  14:36:20        0.003       0.000       0.003       0.001
     4  01-10-85  14:36:20        0.003       0.000       0.004       0.001
INDEX  1 BUILT
     4  01-10-85  14:36:20        0.004       0.000       0.005       0.001

SORTING INDEX  2
 COUNT      DATE      TIME      CPU MIN    DISK MIN    TOTAL TM      DIFF
     0  01-10-85  14:36:20        0.000       0.000       0.000       0.000
SORT COMPLETE
     4  01-10-85  14:36:20        0.005       0.000       0.005       0.005

BUILDING INDEX   2
 COUNT      DATE      TIME      CPU MIN    DISK MIN    TOTAL TM      DIFF
     0  01-10-85  14:36:20        0.000       0.000       0.000       0.000
     1  01-10-85  14:36:21        0.002       0.000       0.002       0.002
     2  01-10-85  14:36:21        0.002       0.000       0.003       0.001
     3  01-10-85  14:36:21        0.003       0.000       0.003       0.001
     4  01-10-85  14:36:21        0.004       0.000       0.004       0.001
INDEX  2 BUILT
     4  01-10-85  14:36:21        0.005       0.000       0.005       0.001

KBUILD COMPLETE.

OK,
```

Example 3: Building Variable-length Records

If you use variable-length records, supply KBUILD with the length of each data record to be written to the output file. The input file, VARNAMES, contains the following records:

```
39276503889harper, anne          chk412389112 washington st newton ma 02159
37036792406harper, anne          ln7253746518 first st dedham ma 02026
41189264289murray, paul          mc28374646123 orchard rd  manchester nh 03102
39023677386corrado, thomas       sav127356542 maple ave arlington ma 02174
```



The input file VARNAMES was used to build the MIDASPLUS file. Although the output records have different lengths, KBUILD needs to know the maximum input data record length (41 words in this example). Each record in the file contains a number that indicates the output record length for that particular record. In this example, the output record length begins in character position 1 and ends in character position 2.

If the number was in binary (INTEGER*2) form, you would indicate the word number that the entry begins in. The KBUILD dialog and the user responses for this example are:

```
OK, kbuild
[KBUILD rev 19.4.0]

SECONDARIES ONLY? no
ENTER INPUT FILENAME: varnames
ENTER INPUT RECORD LENGTH (WORDS): 41
INPUT FILE TYPE: text
ENTER NUMBER OF INPUT FILES: 1
ENTER OUTPUT FILENAME: varbank
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: 1
ENTER ENDING CHARACTER POSITION IN INPUT RECORD: 2
ENTER STARTING CHARACTER POSITION, PRIMARY KEY: 3
SECONDARY KEY NUMBER: 1
ENTER STARTING CHARACTER POSITION: 12
SECONDARY KEY NUMBER: 2
ENTER STARTING CHARACTER POSITION: 37
SECONDARY KEY NUMBER:  (CR)
IS FILE SORTED? no
ENTER LOG/ERROR FILE NAME:  (CR)
ENTER MILESTONE COUNT: 1
```

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

PROCESSING FROM:  varnames
 COUNT      DATE      TIME      CPU MIN    DISK MIN    TOTAL TM      DIFF
     0  01-11-85  14:56:28        0.000       0.000       0.000     0.000
     1  01-11-85  14:56:28        0.002       0.000       0.002     0.002
     2  01-11-85  14:56:28        0.003       0.000       0.003     0.001
     3  01-11-85  14:56:28        0.003       0.000       0.003     0.001
     4  01-11-85  14:56:28        0.004       0.000       0.004     0.001
FIRST BUILD/DEFER PASS COMPLETE.
     4  01-11-85  14:56:29        0.005       0.000       0.005     0.001

SORTING INDEX 0
 COUNT      DATE      TIME      CPU MIN    DISK MIN    TOTAL TM      DIFF
     0  01-11-85  14:56:29        0.000       0.000       0.000     0.000
SORT COMPLETE
     4  01-11-85  14:56:29        0.005       0.000       0.005     0.005

BUILDING INDEX 0
 COUNT      DATE      TIME      CPU MIN    DISK MIN    TOTAL TM      DIFF
     0  01-11-85  14:56:29        0.000       0.000       0.000     0.000
     1  01-11-85  14:56:29        0.001       0.000       0.001     0.001
     2  01-11-85  14:56:29        0.001       0.000       0.001     0.001
     3  01-11-85  14:56:29        0.002       0.000       0.002     0.001
     4  01-11-85  14:56:29        0.003       0.000       0.003     0.001
INDEX 0 BUILT
     4  01-11-85  14:56:29        0.004       0.000       0.004     0.001

SORTING INDEX  1
 COUNT      DATE      TIME      CPU MIN    DISK MIN    TOTAL TM      DIFF
     0  01-11-85  14:56:29        0.000       0.000       0.000     0.000
SORT COMPLETE
     4  01-11-85  14:56:29        0.004       0.000       0.004     0.004

BUILDING INDEX   1
 COUNT      DATE      TIME      CPU MIN    DISK MIN    TOTAL TM      DIFF
     0  01-11-85  14:56:30        0.000       0.000       0.000     0.000
     1  01-11-85  14:56:30        0.002       0.000       0.002     0.002
     2  01-11-85  14:56:30        0.002       0.000       0.003     0.001
     3  01-11-85  14:56:30        0.003       0.000       0.003     0.001
     4  01-11-85  14:56:30        0.003       0.000       0.004     0.001
INDEX  1 BUILT
     4  01-11-85  14:56:30        0.004       0.000       0.005     0.001

SORTING INDEX  2
 COUNT      DATE      TIME      CPU MIN    DISK MIN    TOTAL TM      DIFF
     0  01-11-85  14:56:30        0.000       0.000       0.000     0.000
SORT COMPLETE
     4  01-11-85  14:56:30        0.004       0.000       0.004     0.004

BUILDING INDEX   2
 COUNT      DATE      TIME      CPU MIN    DISK MIN    TOTAL TM      DIFF
     0  01-11-85  14:56:30        0.000       0.000       0.000     0.000
```

```
      1  01-11-85  14:56:30      0.002      0.000      0.002      0.002
      2  01-11-85  14:56:30      0.002      0.000      0.002      0.001
      3  01-11-85  14:56:30      0.003      0.000      0.003      0.001
      4  01-11-85  14:56:30      0.003      0.000      0.003      0.001
INDEX  2 BUILT
      4  01-11-85  14:56:30      0.004      0.000      0.004      0.001

KBUILD COMPLETE.

OK,
```

Example 4: Using Direct Access Files

This section concerns only users with direct access MIDASPLUS files. Direct access MIDASPLUS files are called RELATIVE files in COBOL and DIRECT files in VRPG.

Building Direct Access Files: Building direct access MIDASPLUS files is similar to building keyed-index MIDASPLUS files. The major differences are:

- You must supply a record number for each record. The data type of the record number must be either a REAL*4 (floating-point) number in the form of a bit string, or an ASCII string.
- You must place the record number at the same character position in each input record.
- The record number must be the primary key in COBOL and VRPG files.
- You can supply a primary key in addition to a record number in non-COBOL files.
- A direct access file can have up to 999,999 entries.
- The relative record number cannot be greater than the number of entries allocated during CREATK.

KBUILD Dialog Requirements: After determining that the MIDASPLUS output file is a direct access file, KBUILD 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):
```

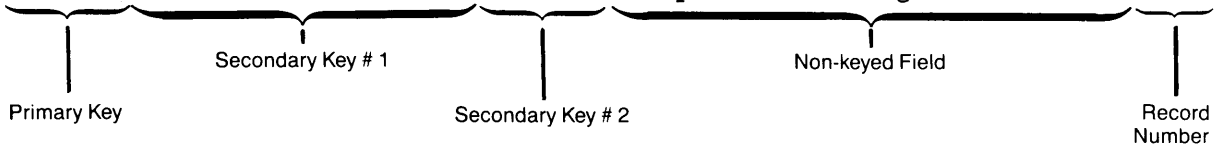
If the record number is an ASCII string, KBUILD prompts for the record number's beginning and ending character positions. If you specify the

number as a single-precision floating-point bit string, KBUILD asks for the word-number (not the character position) where the number begins in the input record. KBUILD warns you if the word number is beyond the logical end of the record that you specified earlier in the dialog. See Chapter 6 THE COBOL INTERFACE, for information on COBOL's direct access (RELATIVE) files. See Chapter 9, THE VRPG INTERFACE, for more information about VRPG's direct access files.

Direct Access Example: You can build a direct access file with KBUILD as long as you include record entry numbers for each record in the input file. Write numbers in ASCII or binary (floating point) form. If the primary key was defined as a record number (this always occurs in COBOL and VRPG), make sure that the numbers match the key type specification.

The following input file was used to build a direct access file. The record number was placed at the end of the file so that the number would not be included in the data subfile record. The direct access entry appears in characters 87 to 92 of the input file record. The file is written in ASCII format.

```
276503889harper, anne          chk412389112 washington stnewton        ma02159000001
036792406harper, anne          ln7253746518 first st     dedham         ma02026000002
189264289murray, paul          mc28374646123 orchard rd  manchester     nh03102000003
023677386corrado, thomas       sav127356542 maple ave    arlington      ma02174000004
```



KBUILD processed this file in the sample session that follows:

```
OK, kbuild
[KBUILD rev 19.4.0]

SECONDARIES ONLY? no
ENTER INPUT FILENAME: directnames
ENTER INPUT RECORD LENGTH (WORDS): 47
INPUT FILE TYPE: text
ENTER NUMBER OF INPUT FILES: 1
ENTER OUTPUT FILENAME: directbank
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: 87
ENTER ENDING CHARACTER POSITION IN INPUT RECORD: 92
ENTER STARTING CHARACTER POSITION, PRIMARY KEY: 1
SECONDARY KEY NUMBER: 1
```

```
ENTER STARTING CHARACTER POSITION: 10
SECONDARY KEY NUMBER: 2
ENTER STARTING CHARACTER POSITION: 35
SECONDARY KEY NUMBER:  (CR)
IS FILE SORTED? no
ENTER LOG/ERROR FILE NAME: (CR)
ENTER MILESTONE COUNT: 1

BUILDING: DATA
DEFERRING: 0, 1, 2
PROCESSING FROM:  directnames
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM     DIFF
     0  01-10-85  18:15:05       0.000      0.000      0.000      0.000
     1  01-10-85  18:15:05       0.002      0.001      0.002      0.002
     2  01-10-85  18:15:05       0.003      0.001      0.003      0.001
     3  01-10-85  18:15:05       0.003      0.001      0.004      0.001
     4  01-10-85  18:15:05       0.004      0.001      0.005      0.001
FIRST BUILD/DEFER PASS COMPLETE.
     4  01-10-85  18:15:05       0.005      0.001      0.005      0.001

SORTING INDEX 0
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM     DIFF
     0  01-10-85  18:15:06       0.000      0.000      0.000      0.000
SORT COMPLETE
     4  01-10-85  18:15:06       0.005      0.000      0.005      0.005

BUILDING INDEX 0
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM     DIFF
     0  01-10-85  18:15:06       0.000      0.000      0.000      0.000
     1  01-10-85  18:15:06       0.001      0.000      0.001      0.001
     2  01-10-85  18:15:06       0.002      0.000      0.002      0.001
     3  01-10-85  18:15:06       0.002      0.000      0.002      0.001
     4  01-10-85  18:15:06       0.003      0.000      0.003      0.001
INDEX 0 BUILT
     4  01-10-85  18:15:06       0.004      0.000      0.004      0.001

SORTING INDEX  1
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM     DIFF
     0  01-10-85  18:15:06       0.000      0.000      0.000      0.000
SORT COMPLETE
     4  01-10-85  18:15:06       0.004      0.000      0.004      0.004

BUILDING INDEX   1
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM     DIFF
     0  01-10-85  18:15:06       0.000      0.000      0.000      0.000
     1  01-10-85  18:15:07       0.002      0.000      0.002      0.002
     2  01-10-85  18:15:07       0.002      0.000      0.002      0.001
     3  01-10-85  18:15:07       0.003      0.000      0.003      0.001
     4  01-10-85  18:15:07       0.003      0.000      0.004      0.001
INDEX  1 BUILT
     4  01-10-85  18:15:07       0.004      0.000      0.004      0.001

SORTING INDEX  2
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM     DIFF
```

```
     0  01-10-85  18:15:07      0.000      0.000      0.000      0.000
SORT COMPLETE
     4  01-10-85  18:15:07      0.004      0.000      0.004      0.004

BUILDING INDEX   2
 COUNT      DATE       TIME     CPU MIN   DISK MIN   TOTAL TM      DIFF
     0  01-10-85  18:15:07      0.000      0.000      0.000      0.000
     1  01-10-85  18:15:07      0.002      0.000      0.002      0.002
     2  01-10-85  18:15:07      0.002      0.000      0.003      0.001
     3  01-10-85  18:15:07      0.003      0.000      0.003      0.001
     4  01-10-85  18:15:07      0.003      0.000      0.004      0.001
INDEX  2 BUILT
     4  01-10-85  18:15:07      0.004      0.000      0.005      0.001

KBUILD COMPLETE.

OK,
```

ALTERNATIVES TO KBUILD

Use an application program, rather than KBUILD, to build a MIDASPLUS file if one or more of the following statements are true:

- You lack a preexisting sequential disk file containing data in an easily convertible form.
- It is more work to prepare an existing data file for use with KBUILD than to use ADD1$ (FORTRAN call interface), an add statement in another language interface, or other offline routines.
- You already have an application program (requiring little revision) that handles additions, updates, deletes, and other processes.
- You regularly make changes that could be handled more easily with an application program.

Chapters 5 through 9 cover data entry using the language interfaces.

KBUILD ERROR MESSAGES

The following are KBUILD runtime error messages. If an error is fatal, KBUILD aborts after reporting it. Although files are sometimes damaged in fatal errors, the files are usually still usable. A non-fatal error is a warning only and does not harm the KBUILD process. The record that causes the warning message, however, is not added to the file.

- UNABLE TO REACH BOTTOM INDEX LEVEL

The last level index block could not be located; file is damaged. (Fatal)

- FILE IN USE

The file is not available for KBUILD use. KBUILD must have exclusive access to the file. You are returned to PRIMOS. (Fatal)

- INDEX 0 FULL -- INPUT TERMINATED

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

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

If the maximum number of entries in the secondary index is exceeded, KBUILD aborts. Building of other indexes continues. (Fatal, but file is still okay)

- INDEX 0 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. KBUILD 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 is still okay)

- INVALID DIRECT ACCESS ENTRY NUMBER -- RECORD NOT ADDED

The user-supplied direct access record number is an ASCII string, but it is not legitimate if it contains non-numeric characters. Also, the entry number may be less than or equal to 0, may not be a whole number or may exceed the 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. Do not specify the sorted option during the KBUILD dialog. (Non-fatal)

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

This error occurs when adding secondary index entries to an already populated file. The primary key value that you supplied in the input file was not found in the primary index. (Non-fatal)

- INDEX 0: INVALID KEY -- RECORD NOT ADDED

This error 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 does not allow duplicates. (Non-fatal)

- INDEX { 0 / index-no: } KEY SEQUENCE ERROR -- RECORD NOT ADDED

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

4

Introduction to File Access

This chapter gives an overview of the MIDASPLUS file operations that you can perform in each programming language; summarizes the available file access operations with each language interface; tells you where to find more information on the language interfaces; gives an overview of direct access.

Chapters 5 through 8 and Appendixes G and H use the BANK file, created in the CREATK chapter, to illustrate MIDASPLUS file access. These chapters explain information retrieval, update, and deletion, with the different language interfaces.

ACCESS OPERATIONS

Accessing a file involves the following operations:

- Opening a file for update and/or reading
- Adding a record
- Positioning to a file record
- Reading a record by any key (partial/full)
- Reading the next record in sequence
- Reading the next (sequential) record with the same key

- Locking a record with a read operation
- Updating the current record
- Deleting the current record
- Deleting a record by a key
- Closing the file

LANGUAGE ACCESS

The following chapters/appendixes of this manual discuss the following language interfaces:

Chapter/Appendix	Language
5	FORTRAN
6	COBOL
7	BASIC/VM
8	PL/I
9	VRPG
G	C
H	Pascal

DIRECT ACCESS

The FORTRAN, COBOL, and VRPG interfaces support direct access MIDASPLUS files based on record numbers. Each record in the file has a unique floating-point record number (single-precision) that identifies that record. To get a particular record, give MIDASPLUS the desired record number; the record is found and returned. When using direct access, you must keep track of the correlation between record numbers and record values.

Direct access uses an algorithm to calculate the exact physical location of the record in the file from the specified record number. Direct access files do not support variable-length records.

Direct Access File Structure

CREATK creates a direct access file template in a manner similar to the one in which it creates a template for keyed-index access MIDASPLUS files. The basic differences between keyed-index and direct access MIDASPLUS files are:

- Direct access files require fixed-length records. Supply the record length (data size) in words.
- Each record in a direct access file requires a unique record number. Depending on the language interface used to access the file, the record number might have to be the primary key.
- Direct access files require preallocation of storage space. Estimate the maximum number of entries that will eventually reside in the data subfile. CREATK will allocate enough space to accommodate a file with the number of records that you indicated.

Note

If keys are included in the file template, you can use both the direct access method and the keyed-index access method to access direct access files in FORTRAN.

RUNNING MIDASPLUS WITH PRIMIX

If a MIDASPLUS application executes a PRIMIX fork system call while MIDASPLUS files are open, automatic cleanup is invoked for the child process. All MIDASPLUS files opened by the parent process are inaccessible to the child process. When automatic cleanup completes, the child process can reinvoke MIDASPLUS by making a file-open call. The parent process continues unaffected. (That is, all files opened previously are still accessible to the parent process.)

5
The FORTRAN Interface

This chapter explains how to use the FORTRAN interface to MIDASPLUS. This interface consists of routines callable from any program written in FORTRAN, PL/I, Pascal, C, or COBOL. Both keyed-index and direct access MIDASPLUS files are discussed as well as the communications array, MIDASPLUS flags, $INSERT mnemonics, and the MIDASPLUS subroutines.

COBOL users have the option to use either the FORTRAN interface or the MIDASPLUS interface available through the COBOL language. (See Chapter 6, THE COBOL INTERFACE.)

FTN and F77 (Prime-supported FORTRAN versions) handle calls to MIDASPLUS identically. Although you can access MIDASPLUS files with either version of FORTRAN, be careful with the differences between the two languages. For example, FTN assumes that variables declared as "INTEGER" are INTEGER*2 while F77 assumes that they are INTEGER*4. See the FORTRAN 77 Reference Guide for a summary of the differences between FTN and F77.

THE CURRENT RECORD

In order to perform the correct operation, some MIDASPLUS calls need to know which record is the current 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 that the next operation knows which record to act upon if necessary.

If the next operation is a read-next operation, the file handler must check the location of the current file position so that it can read the proper record. Since the proper record is the one after the record just read, that record becomes the new current record. If, however, the next operation is a call to FIND$, MIDASPLUS does not care which is the current record because MIDASPLUS is required to do an index search to find the requested record.

The current record position information is stored in a 14-word (28-byte) array supplied on each MIDASPLUS call. MIDASPLUS constantly updates and checks the array. The array contains the index location, file position, and the current record location. This array is called the MIDASPLUS communications array.

DIRECT ACCESS IN FORTRAN

FORTRAN's direct access files do not require you to define the record number as a primary or secondary key. Although you may define the record number as the primary or secondary key, MIDASPLUS stores the record number as a single-precision floating-point number. If you do not want the record number to be a key field, define the primary key as some other unique field in the record. You can also define up to 17 secondary indexes during template creation.

If you decide not to make the record number a key field, do not be concerned about it during template definition. The only time you should be concerned about the record number is when you are adding entries to the file. Then, supply a unique record number for each record to be added to the data subfile. MIDASPLUS stores the record numbers in the proper place.

You can also use KBUILD to build (populate) direct access files. Supply record numbers in the same word position in each record.

To access direct access files by record number (instead of by key), use the same basic subroutine calls. In this case, communications array format is slightly different than for keyed-index access.

THE COMMUNICATIONS ARRAY

After a MIDASPLUS file is opened for access, MIDASPLUS uses the communications array to keep track of the current file position. The array stores the following:

- The current record's address
- The current position in the index subfile
- A status code for the operation

- The word number of the located entry in the index subfile
- The data record address

Most FORTRAN/MIDASPLUS file access subroutines use the array as an argument. Formats and use of the array differ for keyed-index access and direct access.

Keyed-Index Array Format

Only the first word of the array is important for keyed-index access users. You may only modify the first word of the keyed-index access array. When you supply the first word, it can be 0, 1, or -1. When MIDASPLUS returns the first word, it contains the status code of the executed operation. Words 2-14 contain index and data record addresses, subfile numbers, and the key's hash value.

Word 1: Input Value: Word 1 is the only word that you may modify. You may set the value to either 1, 0, or -1. Any other value produces an error on any call in which that array is used. If the value is set to 0 or 1, MIDASPLUS will use the current array contents on the call. If the value is set to -1 (which has precedence over the FL$USE flag), MIDASPLUS ignores the contents of the array. Flag usage is discussed later in this chapter.

Word 1: Output Value: MIDASPLUS always uses the first word in the array to return a completion code after an operation is finished. If set to 0 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 a MIDASPLUS error condition code. Error codes are listed in Appendix B, ERROR MESSAGES.

Direct Access Array Format

In direct access, the first five words of the array are important. Table 5-1 shows the complete format of the array as used in direct access. When using the direct access array, supply the proper values for words 2, 3, and 4 of the array.

Table 5-1
Direct Access Array Format

Word No.	Setting	Meaning
1	0 or 1	Array contents. Supplied by user.
2	entry size (in words)	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		Hash value (based on current key value).

$INSERT MNEMONICS

To allow programmers to refer to error codes and key values by mnemonic names rather than absolute values, the SYSCOM directory contains program insert files. Each PARM.K.INS file contains parameters used in the FORTRAN subroutines; each KEYS.INS contains key declarations. You insert these files in column 1 at the beginning of the program. Use the following statements:

```
$INSERT SYSCOM>PARM.K.INS.FTN          (FORTRAN)
$INSERT SYSCOM>KEYS.INS.FTN            (FORTRAN)

$INSERT SYSCOM>PARM.K.INS.PL1          (PL/I)
$INSERT SYSCOM>KEYS.INS.PL1            (PL/I)

#INCLUDE "SYSCOM>PARM.K.INS.CC"        (C)
#INCLUDE "SYSCOM>KEYS.INS.CC"          (C)

%INCLUDE 'SYSCOM>PARM.K.INS.PASCAL'    (Pascal)
%INCLUDE 'SYSCOM>KEYS.INS.PASCAL'      (Pascal)

COPY "SYSCOM>PARM.K.INS.CBL"           (COBOL)
```

MIDASPLUS FLAGS

A flag is a means of specifying options for a particular call. A flag is actually a switch with a bit value of either on or off. You can set

MIDASPLUS flag parameter values to one or more of the MIDASPLUS keys. Options are specified with a set of flag names that are defined in the insert file SYSCOM>PARM.K.INS.FTN. The flag names correspond to single bits of a one-word parameter called flags and are passed to MIDASPLUS in each subroutine call.

The default setting for each flag bit is off. Depending on what you want to do with a particular call, you set certain flags on before a call. To set a flag on, specify the name of that flag in an assignment statement or in place of the flags argument on the actual call. For example:

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

This example tells MIDASPLUS to position to the first index entry for the specified key and to return the user communication array. As a result of the above assignment, the octal values of the flags FL$FST and FL$RET are added. Their sum, which is a single octal value, determines which bits are set off and which are set on in the flag word. All of the bits are initialized in the PARM.K.INS.FTN file, and the bit settings indicate the actions to be taken on the call.

Figure 5-1 lists the subroutines in which the flags can be used, and Table 5-2 lists the bits to which the flags correspond and their meanings when set on or off.

Flag	ADD1$	FIND$	NEXT$	LOCK$	UPDAT$	DELET$	GDATA$
FL$USE	x	x	x	x	x(R)	x	
FL$RET	x	x	x(R)	x(R)			
FL$KEY	x	x	x	x	x		
FL$BIT		x	x				
FL$PLW		x	x				
FL$UKY		x	x				
FL$SEC		x	x				
FL$ULK					x		
FL$FST		x	x				x(R1)
FL$NXT		x	x				x(Rs)
FL$PRE			x				

(R) = Required.
(R1) = Required for first record.
(Rs) = Required for all records after the first record.

Flags for Subroutines
Figure 5-1

Table 5-2
MIDASPLUS Flag Names, Settings, and Meanings

Bit No.	Name	Setting	Meaning
1	FL$USE	on	Uses current copy of array.
		off	Does not use current copy of array.
2	FL$RET	on	Returns entire array for use on subsequent calls.
		off	Returns completion code only, in array (1).
3	FL$KEY	on	On calls to FIND$, NEXT$, and LOCK$, returns primary key with data record. On calls to ADD1$, tells MIDASPLUS not to store the primary key in the buffer, since MIDASPLUS stores the primary key automatically, using the key argument. Used only if primary key is first field in data record.
		off	On calls to FIND$, NEXT$, and LOCK$, does not add the primary key to the beginning of the data buffer. In ADD1$, tells MIDASPLUS to store a copy of the primary key in each internal data subfile record.
4	FL$BIT	on	If the key is a bit string, the call specifies the key size in bits; if the key is ASCII, the call specifies the key size in bytes.
		off	Specifies key size in words (default).

Table 5-2 (continued)
MIDASPLUS Flag Names, Settings and Meanings

Bit No.	Name	Setting	Meaning
5	FL$PLW	on	Positions to next index entry greater than or equal to current or user-supplied entry.
6	FL$UKY	on	Updates user-supplied key field with version stored in the file. Useful in partial key searches.
		off	Does not update user-supplied key field.
7	FL$SEC	on	Returns secondary data instead of data record.
		off	Returns data record read from data subfile.
8	FL$ULK	on	Unlocks data entry only. Do not update it.
		off	Updates data entry and unlocks it.
9	FL$FST	on	Positions to first index entry in subfile.
		off	Positions to first entry that matches current entry or user-supplied key value.
10	FL$NXT	on	Positions to next index entry greater than current entry or user-supplied key value.
		off	Positions to next index entry that matches current entry or user-supplied key value.
11	FL$PRE	on	Positions to previous index entry.
		off	Does not position to the previous index entry.

Notes

On designates that a particular flag is specified in flags. Off (the default) designates that the flag is not specified. Bits 12-16 of the flags parameter must be set to 0, the default setting, at all times. Do not change the flag setting. When you combine flags, some flags have precedence over others. The priority level is:

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

Certain combinations of flags are not sensible. For example, FL$NXT and FL$FST. Although meanings are given when each flag is set off, the combination of keys that are set on is what actually dictates the action.

COMPILE AND LOAD SEQUENCE

You must include the MIDASPLUS library MPLUSLB in the BIND load sequence of all FORTRAN programs that use MIDASPLUS. Substitute the name of your program for the word program in the example.

A sample BIND session using FTN:

```
ftn program -64v
0000 ERRORS [<.MAIN.>FTN-REV19.3]
OK, bind
[BIND rev 19.4]
: load program
: li mpluslb
: li
BIND COMPLETE
: file
OK, resume program
```

A sample BIND session using F77:

```
f77 program -ints
0000 ERRORS [<.MAIN.>F77-REV19.3]
OK, bind
[BIND rev 19.4]
: load program
: li mpluslb
: li
BIND COMPLETE
: file
OK, resume program
```

Note

You must use the option -INTS with F77. F77 defaults to long integer if -INTS is not used.

The FORTRAN/MIDASPLUS INTERFACE SUBROUTINES

A FORTRAN programmer can directly use ten FORTRAN subroutines to access a MIDASPLUS file. The other language interfaces also use these subroutines, though transparently to the user. Most of these subroutines share the same calling sequence. The following is a list of the subroutines and their functions:

Subroutine	Function
OPENM$	Opens a MIDASPLUS file, associates it with a file unit, and notifies MIDASPLUS that processing is about to begin on the file.
CLOSM$	Closes a file and its subfiles.
NTFYM$	States that a MIDASPLUS file is opened or is about to be closed.
ADD1$	Adds a data record and index entries.
FIND$	Finds a data entry by any key.
NEXT$	Finds the next data entry via an index.
GDATA$	Reads data entries in the order stored.

LOCK$	Locks a data entry for update.
UPDAT$	Updates a data entry.
DELET$	Deletes a data entry or secondary index entry.

Record Locking

In order to update a record, lock that record for exclusive use. Locking the record stops anyone else from trying to read or change the record while you are changing it. While FORTRAN requires you to call the LOCK$ subroutine before an update operation can occur, the other MIDASPLUS language interfaces automatically perform locking. Locking prevents other users from updating the record, but will not protect against deleting it.

General Calling Sequence

The MIDASPLUS access subroutines (ADD1$, FIND$, LOCK$, DELET$, NEXT$, and UPDAT$) use the following arguments as a general format for their calling sequence:

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

Note

routine is a character data type, but all of the other arguments are short integer (INT*2).

Argument	Specifies
routine	One of these six routines: ADD1$, FIND$, NEXT$, LOCK$, UPDAT$, or DELET$.
funit	The file unit on which the MIDASPLUS file is open.
buffer	The data record buffer into which data is read or from which it is written to the file.
key	The key value to be used in the call.
array	The communications array that holds current record and index position information. It also returns status codes after each call.

flags	The flag options for this call.
altrtn	The statement label in the program to which control passes if an error occurs. Set to 0 if no alternate return exists.
index	The access method to be used (keyed-index or direct access) and the index subfile number to use if not direct access.
file-no	Ignored by MIDASPLUS, but kept for compatibility with older versions.
bufsiz	The length of the data to be transferred to/from file (except in calls to DELET$). Set to 0, if full data entry is being transferred. Always supplied in words.
keysiz	The length of the key to be used in partial key access (used with FIND$ and NEXT$ only).

Optional Arguments: You may supply a 0 instead of another value for the following arguments:

Argument	Default
altrtn	No alternate return for handling errors on this call.
file-no	Obsolete, maintained for compatibility.
bufsiz	Defaults to data subfile entry length (stored in file).
keysiz	Defaults to key length specified in file. Can be set to 0 if full key is being used.

Note

GDATA$, a data access subroutine used for sequential retrieval of entries in the data subfile, does not use the general calling sequence just described.

OPENING AND CLOSING MIDASPLUS FILES

MIDASPLUS requires that you open or close a file through MIDASPLUS or otherwise notify MIDASPLUS of every MIDASPLUS file that you open or close. You may use either of the following methods to open a file:

- Use the OPENM$/CLOSM$ subroutines.
- Make calls to NTFYM$ to modify existing programs that use SRCH$$ or SRSFX$ to open and close a file. When SRCH$$ or SRSFX$ opens a MIDASPLUS file, make the call to NTFYM$ after opening the file. When SRCH$$ or SRSFX$ closes a MIDASPLUS file, issue NTFYM$ before the closing the file.

OPENM$

OPENM$ is the MIDASPLUS routine that opens a MIDASPLUS file (segment directory). Through OPENM$, MIDASPLUS opens a file, validates it as a MIDASPLUS file, and stores the information in its file table. MIDASPLUS requires a call to OPENM$ (or NTFYM$ as an alternative but unequal choice) before a file can be accessed using the online MIDASPLUS routines. If you try to access a file that MIDASPLUS is unaware of, an error code of 23 will appear saying that the file is not opened.

Always use the same access mode when opening a file more than once from the same program. To manage file units more efficiently, open a MIDASPLUS file only once in the same application.

OPENM$ Keys

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

- SRCH$$ (takes a filename argument)
- SRSFX$ (takes a pathname argument)

These routines open a file and associate the file with a PRIMOS file unit. OPENM$ requires the use of certain PRIMOS keys (listed below) that specify whether to open a file for reading, writing, or a combination of both:

Key	Action
K$GETU	Opens a file on an available PRIMOS file unit.
K$READ	Opens a file for reading only.
K$WRIT	Opens a file for writing only.
K$RDWR	Opens a file for reading and writing.

Note

The keys listed above are used in calling OPENM$ as shown below. You must specify one of the choices K$READ, K$WRIT, or k$RDWR. K$GETU is strongly recommended.

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	Input parameter. Valid OPENM$ access key: either K$READ or K$WRIT or K$RDWR used together with K$GETU.
pathname	Pathname of MIDASPLUS file to be opened.
namlen	Length of the pathname in characters, supplied by the user. F77 programs will not run if the namlen parameter is supplied as a constant.
funit	The file unit on which the file was opened, returned by OPENM$ when K$GETU is specified.
status	OPENM$ status code returned by MIDASPLUS at the completion of the call. Possible values are

0	No error.
< 10001	PRIMOS file system error.
10001	Invalid key supplied.
10002	Too many MIDASPLUS files are open. The default is 256 file units and the maximum is 512.
10003	Specified file is not a MIDASPLUS segment directory.
10004	Internal error. Ask the System Administrator for assistance.
10005	Internal error encountered while trying to open a remote file.

CLOSM$

The CLOSM$ routine closes a MIDASPLUS file (segment directory) that was opened on a specified file unit. CLOSM$ also closes any of the subfiles that MIDASPLUS opened during file access. CLOSM$ can also be called as a function, returning these values in the status code return argument:

0	No error.
1	Error occurred. Check status code.
-1	File was not opened and status code is 0.

CLOSM$ Calling Sequence

The calling sequence of CLOSM$ is:

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

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

funit	Input parameter. File unit on which the MIDASPLUS file is opened.
status	Output parameter. CLOSM$ status codes.

0	No error.
< 10001	PRIMOS file system error.
10001	MIDASPLUS is unaware that the file is opened. Internal error. Ask the System Administrator for assistance.

NTFYM$

Use NTFYM$ if you are using SRCH$$ to open files rather than OPENM$. The NTFYM$ routine informs MIDASPLUS that you either opened a MIDASPLUS file (segment file) or are about to close a file using SRCH$$ or SRSFX$. Place NTFYM$ into an existing program immediately after a call to SRCH$$ is made to open the file and immediately before SRCH$$ is called again to close the file.

Before a MIDASPLUS file is closed, a call to NTFYM$ tells MIDASPLUS to close any of the file's segment subfiles that were left open. MIDASPLUS requires a call to OPENM$ or NTFYM$ before you can access a file with the online MIDASPLUS routines. Without this call, the online routines return an error code of 23 (File not Found).

NTFYM$ Calling Sequence

The calling sequence is:

```
CALL NTFYM$ (key, funit, 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 (user supplied).

- 1 = file is open
- 2 = file is about to be closed

funit — File unit on which MIDASPLUS file is opened (user-supplied).

status — NTFYM$ status codes

0	No error.
< 10001	PRIMOS file system error.
10001	Invalid key supplied.
10002	Too many MIDASPLUS files are opened. The default is 256 and the maximum is 512.

10003	Specified file is not a MIDASPLUS segment directory.
10004	Internal error. Ask the System Administrator for assistance.
10005	Internal error encountered while trying to open a remote file.

ADD1$

Use the ADD1$ routine to add primary index entries and data subfile entries to keyed-index and direct access MIDASPLUS files. You can add secondary index entries (and optional secondary data) to files with secondary indexes. If you want your secondary keys to be in the data record, make sure that the secondary keys exist in the buffer when the primary key and record are added to the file.

If ADD1$ adds a variable-length record that is outside a record size limit, MIDASPLUS automatically resets that limit to the size of the record.

Note

Because MIDASPLUS is a word-aligned product, the key or record buffer passed to MIDASPLUS must be word-aligned. In this manual, the term word means 16 bits.

Keyed-index Adds

You can only add records to a MIDASPLUS file when the records have a primary key value. Likewise, you can only add a secondary key value if the record which it will reference already exists in the data subfile and if a primary index entry references it. Add records and keys associated with them in the following order:

1. To add a data entry and its primary value, make a call to ADD1$ with the flag FL$RET set on.

2. To add a secondary index entry for this record, make a separate call to ADD1$ with the flag FL$USE set on.

In order to add secondary index entries for an existing data subfile at a later time, you can do either one of the following:

- Supply the primary key value in the argument buffer and set index and key to the desired secondary index number and value, respectively, on a call to ADD1$.

- Use the primary key (with a FIND$ or NEXT$ call using FL$RET) to locate the record. This call returns the array so that the call to ADD1$ can use it. Set the index and key to the index number and value, respectively, on the call to ADD1$.

ADD1$ Calling Sequence

The calling sequence format is:

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

Table 5-3 explains the arguments that have meanings unique to ADD1$.

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

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

When index is 0, use the buffer to supply the data entry information which will be added to the data subfile. If you are storing keys in the secondary indexes, make sure that index is a number from 1 to 17. For secondary keys, specify the corresponding primary key as the first item in the buffer and specify the secondary value in the full key argument.

Table 5-3
ADD1$ Arguments

Argument	Meaning
funit	File unit on which the MIDASPLUS file is open.
buffer	Buffer containing the data subfile record on a data entry add operation. If keys are being stored in the data record, include all key values in buffer as well. On a secondary index add without FL$USE set, buffer contains the primary key value followed by optional secondary data.
key	Value of the key that must contain full primary key value on a data record add. Contains full secondary key value on a secondary index add.
array	Communications array that returns a completion or error code.
flags	The switch with a bit value that can be set either on or off. (See Table 5-4 for flags that can be used with ADD1$.)

Table 5-3 (continued)
ADD1$ Arguments

Argument	Meaning
altrtn	Statement number of alternate return to be used in case an error occurs on the subroutine calls. Supply 0 if you cannot use an alternate return.
index	Access method and index subfile to use. The values for the index argument are listed in the Index Value section above.
file-no	Set to 0 (obsolete).
bufsiz	Length of data to be added to subfile from buffer. If bufsiz = 0 and index = 0, MIDASPLUS adds data from buffer to data subfile. MIDASPLUS takes only the number of words it needs to match the record size defined for the MIDASPLUS file during CREATK. If index > 0 and secondary data is supported for that index, adds secondary data from buffer to indicated index subfile. 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 the data subfile or secondary data entry is zero-filled. In general, specify 0 for fixed-length records. For files with variable-length records, specify length of data to be written to file.
keysiz	Set to 0 (ignored).

Table 5-4
Flags for ADD1$

Flag	Function
FL$USE:	Uses contents of the array from previous call - used on calls to add secondary index entries. Set off when adding primary index entries or secondary index entries if using the primary key to locate data records.
FL$RET:	When set on, returns array contents from this call (set only on calls to add data records).
FL$KEY:	When set on, tells MIDASPLUS that the primary key is included at the beginning of the buffer. Since the primary key is already stored with the data, this flag alerts MIDASPLUS that the data portion of the record is offset from the start of the buffer by the length of the primary key. For storage purposes, MIDASPLUS will ignore the primary key portion of the buffer.

Redundant Primary Keys

Since MIDASPLUS always stores a copy of the primary key along with the data record, redundant primary keys would result if the file were created with the primary key as part of the data record. However, users may wish to consider the primary key as part of the data record. To solve this problem, MIDASPLUS provides a flag, FL$KEY. FL$KEY tells MIDASPLUS to place the primary key in the front of the record buffer argument for retrieval operations and to ignore the primary key in the beginning of the buffer on update and insert operations. Logically, the user will see the key as part of the data record. Physically, MIDASPLUS will store only one copy of the key.

Adding Data Records

When adding primary index and data entries, place the full primary key value associated with the record in key. Place the information added to the data subfile in the buffer. Set bufsiz to 0 for keyed-index MIDASPLUS files with fixed-length records. Set bufsiz to the length of the data entry in words for variable-length records. If you are adding secondary indexes to this file, return the array (set the FL$RET flag on in flags) for use in later calls to ADD1$.

Note

When adding entries to a MIDASPLUS file with ADD1$, supply the full key value in the key argument. Partial key values are illegal. Since the argument keysiz is ignored, set it equal to 0.

Adding Secondary Index Entries

To add secondary index values, supply MIDASPLUS with the following information:

- Secondary index number (in index).
- Secondary key value (in key).
- Primary key value -- place it in the first part of buffer or set FL$USE to use a valid copy of array. The array is valid only if the previous call (in which the desired key value was used and/or returned) returns it.
- Secondary data (optional - supplied in buffer following primary key value).

Duplicate secondary key entries are supported only for those index subfiles that were created with duplicate status during CREATK. If you try to add duplicate entries to a secondary index that does not support them, an error is returned and the add operation fails.

For example, if you want to add all secondary index entries for a particular data subfile entry, perform the following eigth steps immediately after you have added the primary key and the data entry:

1. Set FL$RET in flags on the ADD1$ call when adding the primary index and data entry.
2. If you have one or more secondary index entries to add after the above call, set FL$USE in flags.
3. Set index to the appropriate index subfile number (1 - 17).
4. If the array is not valid, put the primary key value of this record in the first part of the buffer.
5. If adding any secondary data for this index entry, put the secondary data in buffer immediately following the primary key value.
6. Set key to the full secondary value that you want stored in the index subfile.
7. Call ADD1$ to add this secondary index entry.

8. Repeat steps 2 - 7 for each secondary index entry that you want added for this record.

Direct Access Adds

For direct access files, set the index value to -1 in all calls to ADD1$ that add data entries. To add secondary index entries, set index to the secondary index subfile. The record number may be defined as the primary key. Since MIDASPLUS stores the numbers, it is not necessary to define the record number as a key during the template creation.

Provide the following in each call to ADD1$:

- A primary key value (in key)
- A floating-point entry number (in words 3 and 4 of the array)
- The data entry size (in word 2 of array)

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. For example, given a primary key size of 3 words and a data entry size of 10 words, the data size argument would be 15.

The Array: See Table 5-1 for the array format for direct access calls. The contents of the array's first four words in direct access calls to ADD1$ are:

- Word 1: Condition code (0 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, MIDASPLUS places the new record number into an overflow area. Duplicates are not allowed for any primary key. Therefore, MIDASPLUS will not place the new record into an overflow area if the primary key is defined as the record number.

Return Code Values

Common return codes in word 1 after a call to ADD1$ are:

Code	Meaning
0	Successful completion of the call.
1	Successful completion of the call. There are duplicates for this index (okay).
7	No entry exists with supplied primary index value.
12	You attempted to add duplicates to a primary index or to a secondary index that does not allow them.
other codes	See Appendix B, ERROR MESSAGES, for a list of MIDASPLUS error codes.

READING A MIDASPLUS FILE

Use FIND$ or NEXT$ to perform keyed reads of primary and secondary keys. For direct access files, you can use FIND$ to read by record number. Use GDATA$ to retrieve records directly from the data subfile in the order in which they appear.

Note

Unlike some of the other language interfaces, none of the FORTRAN interface's data retrieval routines locks a record upon positioning to it. To lock a record, use LOCK$.

FIND$

Using either a primary or a secondary key, FIND$ locates and reads a MIDASPLUS data entry. Searches can also be done on partial primary or secondary key values. If a partial key search is used, you can request FIND$ to return the full key value as stored in the index subfile being searched. If a secondary index contains secondary data, you can request FIND$ to return the secondary data instead of the data record.

FIND$ Calling Sequence

The calling sequence for FIND$ is:

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

Table 5-5 shows the arguments that have special meaning. Table 5-6 lists the flag values valid for use with FIND$.

Specifying Which Index to Use

The index argument tells which access mode is being used on this call to FIND$. It also states which index will be used in a keyed-index file. The settings are:

Index Values	Access Mode
0	Use primary index as search key.
1-17	Use indicated secondary index as basis of search.
-1	Direct access: locate entries by record number.

Read the FIND$ and Direct Access section in this chapter for information on reading direct access files.

Table 5-5
FIND$ Arguments

Argument	Meaning
funit	File unit on which the MIDASPLUS file is open.
buffer	Buffer in which data entry, primary key values, or secondary data are returned as a result of call to FIND$.
key	Full or partial primary or secondary key value supplied by the user. In direct access, if index=-1, do not supply a value for key unless FL$UKY is set in flags.
array	The communications array that returns a completion or error code in word 1 after each call. In direct access, you are required to supply the entry size and record number in words 2-4 of this array.
flags	The switch with a bit value that can be set on or off. (See Table 5-6 for flags that can be used with FIND$.)
altrtn	Statement number of alternate return to be used in case an error occurs on the subroutine calls. Supply 0 if you cannot use an alternate return.
index	Access method and index subfile to use. The values for the index argument are listed in Specifying Which Index to Use above.
file-no	Obsolete, set to 0.
bufsiz	Length of buffer. Set to 0 if complete data entry and primary key (if FL$KEY is set on) are to be returned. Otherwise, specify the number of words you want returned from data subfile entry.
keysiz	Size of key. Set to 0 if full key, otherwise set to number of bits, bytes, or words in key. The keysize is assumed to be in words unless FL$BIT is on, in which case keysize is bits or bytes depending on the key type.

Table 5-6
Flags for FIND$

Flag	Meaning
FL$BIT	Specifies that the length of key in keysiz is in bits if the key is defined as a bit string, or in bytes if the key is an ASCII string. When set off, length of key is specified in keysiz in words.
FL$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$KEY	Tells FIND$ 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$NXT	Positions to the next index entry that is greater than the current value. When set off, FL$NXT positions to the index entry that matches the current or user-supplied entry. A flag setting with a higher precedence (FL$FST) may overrule this.
FL$PLW	Positions to meet the index entry that is greater than or equal to the current or user-supplied value.
FL$RET	Tells MIDASPLUS to return entire array after this call to FIND$. If set off, only the first word of array, the completion code, is returned.
FL$SEC	Returns secondary data from the secondary index being searched instead of returning the data record. Secondary data is returned in buffer. Not applicable in direct access or primary index access, that is, if index is 0 or -1.
FL$UKY	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, tells MIDASPLUS to use contents of array as returned by previous call. In direct access, the setting is ignored.

Specifying Key Values

If index is 0, supply a primary key in the key argument. If you use FL$USE, however, the key is not needed. If a secondary index is indicated, specify a value from the index in key. This allows FIND$ to use it in the retrieval. If the full key value is specified in key, you can set the keysize argument to 0.

Partial Key Values: You may use full or partial keys in both primary and secondary key searches. Partial key values must be the left-most characters of the key (i.e., take the values from the beginning of the key value). For example, if the key value is Massachusetts, possible partial values include M, Ma, Mass, Massach, 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 indicate a partial key value, supply, in keysiz, the exact length of the value you have specified in key. If the key is a bit string or an ASCII string, specify the key's length in bits or bytes as appropriate, and set the FL$BIT flag on. If FL$BIT is set off, the key size is assumed to be in words. How the file was created determines what the value of FL$BIT should mean. If its binary form is used, bits are assumed with FL$BIT. If ASCII is used, bytes are assumed.

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

Retrieval Options

FIND$ permits you to retrieve the following items from a MIDASPLUS file:

- A data subfile (full or partial). Set bufsiz to 0 to return all information. To return a partial entry, specify in bufsiz the number of words that you want returned.

- The primary key value associated with the record that is sought. Set FL$KEY on. Use this method when keys are not stored in the data record, or when entries were added with FL$KEY set on during calls to ADD1$. Primary key value along with the data record 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 conducted. Secondary data is returned in buffer in place of a data record. Set bufsiz to 0 to return all secondary data, or to the number or words that you want returned. Use FL$SEC in the call. (Index: must be a value between 1 and - 17.)

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

All information that FIND$ returns is placed in buffer (except when you specify FL$UKY to return key value in key). Set bufsiz to accommodate all of the data during each call.

Using FL$KEY: It is only necessary to set FL$KEY on (to return full primary key value) when FL$KEY was set on during calls to ADD1$ or when you are not storing keys in the data record.

Using FL$UKY: The FL$UKY flag is useful when you are doing record access via partial key. FL$UKY returns the complete value of the key that was used for the search. If you store keys in the record, you can check to make sure that the index entries correspond to the key values in the data record. These keys also help identify which record you are looking at when you are doing retrievals on duplicate keys.

FIND$ and the Array

During access to keyed-index access MIDASPLUS files, you do not have to worry about the settings for the array argument in calls to FIND$. Word 1 always gives you a completion code after a call to FIND$. If the value of array(1) is 0, the call was successful. A value of 1 indicates that there are duplicates.

When the FL$RET flag is set in the call to FIND$, the entire array is returned to you. You can subsequently use the array in calls to other routines such as ADD1$, NEXT$, DELET$, and LOCK$.

FIND$ and Direct Access

You can access direct access files by any key or entry number. To access a direct access file by primary or secondary key, use the keyed-index access method. Treat the direct access file just like a keyed-index access file. 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 0. In some direct access files, the entry number and the primary key must 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 the floating-point data entry number in array words 3 and 4 and the full data subfile entry in bufsiz, in words.

Argument Settings: Set the index argument to -1 for accessing by entry number. If you use the primary key to search, the key must contain the full primary key value used in the search, and keysiz must always be 0 (indicating a full key value). For the direct access method, set the array argument to the entry number used on the call. Since MIDASPLUS always uses the array on this type of call, do not set FL$USE for this type of call. See Table 5-1 for the direct access array format.

NEXT$

NEXT$ allows you to perform a variety of operations on a keyed-index access MIDASPLUS file. Use NEXT$ to retrieve the following:

- File records sequentially according to primary or secondary key order
- All file records with a primary or secondary key value greater than a given key value
- All records with the same partial key value
- All records with duplicate index entries for a specified secondary key value
- All records whose key values come before a certain key value in a particular index subfile
- A particular record using a full or partial primary or secondary key value (keyed retrieval)

See Table 5-8 for the special flag settings required to perform these retrievals.

Note

Because you cannot use NEXT$ on direct access files, index will never have a value of -1 in a call to NEXT$. Specify FL$RET in calls to NEXT$ or a MIDASPLUS error will occur.

NEXT$ Calling Sequence

The calling sequence for NEXT$ is:

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

See Table 5-7 for the meanings of NEXT$ arguments and Table 5-8 for the flag arguments settings for NEXT$.

Table 5-7
NEXT$ Arguments

Argument	Meaning
funit	File unit on which the MIDASPLUS file is open.
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 5-8.
key	Value of the key used in the search. Either full or partial, as specified in keysiz.
array	Communications array that returns a completion or error code.
flags	Switch that can be set on or off. (See Table 5-8 for flags that can be used with NEXT$.)
altrtn	Statement number of the alternate return to be used in case an error occurs on the subroutine calls. Supply 0 if you cannot use an alternate return.
index	Index subfile to use. Direct access is illegal (index cannot be -1). 0 = primary index 1-17 = secondary index
file-no	Obsolete, set to 0.
bufsiz	Length of data to be returned. If set to 0, full data subfile entry is returned. If FL$KEY is set on, the full key value is returned with the data. If FL$SEC is set on, secondary data is returned instead of the data subfile entry. Make the value of bufsiz large enough to accommodate everything that must be returned in buffer.
keysiz	Length of user-supplied key on this call. If set to 0, full key value is used. If greater than 0, partial key is specified in either bits or bytes (if FL$BIT is set on) or in words (FL$BIT set off).

Table 5-8
Flags for NEXT$

Flag	Function
FL$BIT	Specifies that the keysiz is specified in bits or bytes. When set off, keysiz is in words.
FL$FST	Tells NEXT$ to return the record referenced by first entry in the specified index.
FL$KEY	Tells NEXT$ 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$NXT	Positions to next index entry greater than key.
FL$PLW	Positions to next index entry greater than or equal to key.
FL$RET	Tells MIDASPLUS to return the contents of the array after this call. This flag is required on calls to NEXT$. If it is set off, an error code of 30 will appear.
FL$SEC	When set on, FL$SEC returns secondary data in buffer instead of data record. Use FL$SEC only when index is greater than or equal to 1.
FL$UKY	Returns in key the full primary or secondary index value that corresponds to the user-supplied key value used in this call.
FL$USE	Tells MIDASPLUS to use the contents of array. (Array must be present from a previous call to FIND$ or NEXT$.)
FL$PRE	Finds the previous index entry when the array is used for positioning.

Buffer Size Specifications

Data retrieved on a call to NEXT$ is returned in buffer. The bufsiz argument determines the amount to be returned. To return the entire data subfile entry, set bufsiz to 0. Also set bufsiz to 0 when retrieving secondary data (when index is set to a value greater than 0 and FL$SEC is set). Otherwise, set this argument to the number of words that you want returned from the index or data subfile. Make sure

bufsiz specifies a large enough buffer to include the full primary key and the data record when FL$KEY is used.

Array Settings

Word 1 returns a completion code after the call. The settings for array(1) are:

Code	Meaning
0	Successful retrieval.
1	Successful retrieval, but duplicate may exist for this key value.
other codes	Error in retrieval. See Appendix B, ERROR MESSAGES, for a list of the MIDASPLUS error codes.

Sequential Record Retrieval

To retrieve records sequentially from some point in a primary or secondary index, use FIND$ to locate the initial key value. Once the starting point is found, make repeated calls to NEXT$ to return the data subfile records based on the order of entries in the primary or secondary index. In combination, FIND$ and NEXT$ calls enable a "greater than or equal to" search. First, you find a particular value; then you find all of the values that are greater than or equal to it.

To start this type of retrieval, set the FIND$ call flag to FL$RET so that you can use the returned array in the NEXT$ loop. Set the FL$RET and FL$USE flags in the NEXT$ call. The array that the FIND$ call returns is used for the first NEXT$ call. The NEXT$ call returns another array, which acts as input for the following NEXT$, and so forth.

To retrieve file records sequentially, begin with the first index entry in a given subfile and 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 next call. This tells NEXT$ to get the next entry in the index regardless of whether it matches the one just retrieved or not.

Retrieving Duplicates

NEXT$ can retrieve duplicate secondary key values or key values that begin with identical prefixes. Take key values from the first part of the full key value. For example, if the full key is Brookline, acceptable prefixes 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$PLW are set off.) Set the FL$RET flag on for all calls to NEXT$ when doing this type of retrieval.

GDATA$

GDATA$ is used for sequential access only and retrieves records directly from the data subfile in the order that they appear in the data subfile. Unless the records were added in order by primary key or an MPACK was performed on the file by DATA, this order does not necessarily correspond to any key order.

Set the FL$FST flag on in the first call. Set the FL$NXT flag on in the following calls.

> **WARNING**
>
> Successive calls to GDATA$ with FL$NXT cannot be mixed with calls to other MIDASPLUS file access routines. Only use GDATA$ on one file at a time.

GDATA$ Calling Sequence

The unique calling sequence for GDATA$ is:

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

See Table 5-9 for the file arguments and their meanings.

Table 5-9
GDATA$ Arguments

Argument	Meaning
funit	PRIMOS file unit on which a MIDASPLUS file is open.
flags	The record to be retrieved. For the first call, set it to FL$FST to retrieve the first record in the data subfile. For subsequent calls, set it to FL$NXT to retrieve the next sequential record.
buffer	Buffer in which data is returned.
bufsiz	Size of buffer in characters.
status	Error codes include: 0 No error >0 System error code -1 Bad flag value supplied -3 Invalid record position -4 Fatal internal error

After returning from a successful GDATA$ call buffer contains the retrieved data record.

UPDATING A RECORD

LOCK$ and UPDAT$ are used together to perform a record update. LOCK$ secures a record for update and prevents other users from locking or updating the record. Other users may still delete a locked record. To update a record, lock the record with LOCK$ and then update it with UPDAT$.

LOCK$

LOCK$ works on both keyed-index and direct access MIDASPLUS files. It is similar to FIND$ except that LOCK$ also locks the record it retrieves. LOCK$ returns the located data record in buffer. LOCK$ cannot lock an already locked record, and returns an error if you try to do so. When LOCK$ is successful, the record remains locked until you call UPDAT$ to update or unlock the record. UPDAT$ is the only way to unlock the record. Always call UPDAT$ after a successful call to LOCK$.

LOCK$ Calling Sequence

The LOCK$ calling sequence is:

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

See Table 5-10 for an explanation of the arguments and Table 5-11 for a list of the LOCK$ flags.

Table 5-10
LOCK$ Arguments

Argument	Meaning
funit	File unit on which the MIDASPLUS file is open.
buffer	Buffer into which the data record to be locked is read.
key	Full primary or secondary key value that identifies the record to be locked. This argument is not necessary if the record was already retrieved by a call to FIND$ or NEXT$.
array	Communications array that returns a completion or error code after each call. For direct access, array must include the user-supplied record number and size to identify the record to be locked. See LOCK$ and Direct Access below.
flags	Switch that can be set either on or off. (See Table 5-11 for flags that can be used with LOCK$.)
altrtn	Statement number of alternate return to be used in case an error occurs on the subroutine calls. Supply 0 if you cannot use an alternate return.
index	Access method and index subfile to use: 0 = primary index 1-17 = secondary index -1 = direct access
file-no	(Obsolete, set to 0.)
bufsiz	Length of the data to be read from the file. Set it to 0 to return the whole record. If FL$KEY is set on, make sure that bufsiz is large enough to include the primary key along with the record buffer.
keysiz	(LOCK$ ignores this. Full key is assumed if supplied.)

Table 5-11
Flags for LOCK$

Flag	Function
FL$KEY	When set on, includes full primary key value in buffer along with the data record. Use this flag only if keys are not stored in the record.
FL$RET	Required in each call to LOCK$. This flag allows UPDAT$ to use the array that this call returns. If not set, an error 30 occurs.
FL$USE	Set on if the previous call to FIND$ or NEXT$ already found the record that was to be locked. The previous call's returned array is used on this call to LOCK$. You do not have to supply a value for key.

Specifying a Key

In order to lock a record, you must retrieve it and make it current. To retrieve the record to be updated, supply a full key value in key on a call to LOCK$. Otherwise, make sure a valid array has been supplied by a previous call and that FL$USE is set. You can use the primary or secondary key to position and lock the record.

Partial key retrieval is possible only if you use FIND$ or NEXT$ first with FL$UKY and FL$RET set on in flags. You can then call LOCK$ with FL$USE set on. No key is required in this call to LOCK$ because the previous FIND$ or NEXT$ call has already located the data record.

The Array in LOCK$

When you have already found the entry to be updated on a previous call to FIND$ or NEXT$ (with FL$RET set on), set FL$USE on in flags on the call to LOCK$. The first word of the array (the completion code) may contain one of the following values.

Value	Meaning
0	Successful retrieval.
1	Successful retrieval. There might be duplicates of this key value (secondaries only).
7	Entry not found.
10	Entry found, but already locked.
other error code	See Appendix B, ERROR MESSAGES for a list of MIDASPLUS error codes.

Note

On all calls to LOCK$, set FL$RET on so that the next call to UPDAT$ can use the array that the LOCK$ operation returns.

LOCK$ and Direct Access: The use of LOCK$ with direct access resembles the use of LOCK$ with keyed-index access. Unlike keyed-index access, set the index to -1 with direct access. If a prior call to FIND$ does not return an array, include the data entry number and size in the array. Set up the array as follows:

Word Number	Setting
1	If set to 1, the array contents are used. If set to -1, the 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 0 (obsolete).

To retrieve records before locking them, call FIND$ with FL$RET set on. Then call LOCK$ with FL$USE set on. It is not necessary to reset the array.

UPDAT$

Always call LOCK$ before calling UPDAT$. After the LOCK$ call, check the returned completion code in array(1) to make sure that the record was successfully locked before calling UPDAT$. Record updates are allowed on both keyed-index and direct access MIDASPLUS files. An update is a true rewrite of the record as returned in buffer. After the UPDAT$ call, the record is unlocked. To unlock the record without updating it, call UPDAT$ with FL$ULK set on.

UPDAT$ Calling Sequence

The calling sequence for UPDAT$ is:

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

Since key values are not supplied in updates, both the key and keysize arguments should be set to 0 in a call to UPDAT$. Index must match index specified on the prior call to LOCK$. The updated record is supplied in buffer. Table 5-12 describes the UPDAT$ arguments and Table 5-13 describes the UPDAT$ flags.

Unlock only: If you want to unlock a record without updating it, set FL$ULK on in flags. It is not necessary to change the buffer; the record will not be rewritten.

UPDAT$ and the Array: Always set the FL$USE flag on when calling UPDAT$. The array is supplied by FL$USE being set on and the array should not be tampered with following a call to LOCK$. The completion code indicates whether the update was successful. The update was successful if array(1) is returned as 0. If the completion code was returned as 11, the entry was not locked and the operation failed. Other errors also occur, such as a concurrency error if another user deletes the record between LOCK$ and UPDAT$ calls. See Appendix B, ERROR MESSAGES, for a list of MIDASPLUS error codes.

Note

Neither primary nor secondary keys including secondary data values can be changed in a call to UPDAT$. When keys are stored in the data record, changes to secondary key fields (during a call to UPDAT$) will not affect the secondary index subfile entries that point to the updated record. To change a secondary index entry and/or secondary data, delete the entry from the index subfile, and then re-add it in the desired

manner. If the keys are stored in the data record, you should then update the data record accordingly.

Table 5-12
UPDAT$ Arguments

Argument	Meaning
funit	File unit on which the MIDASPLUS file is open.
buffer	Buffer that contains the record as it is to be rewritten. If FL$KEY was set in the previous call to LOCK$, include the primary key in buffer.
key	(Ignored, set to 0.)
array	Communications array that the previous call to LOCK$ supplies. The array should, if successful, already be set to 0 or 1.
flags	Switch that can be set either on of off. (See Table 5-12 for flag options on update calls.)
altrtn	Statement number of alternate return to be used in case an error occurs on the subroutine calls. Supply 0 if you cannot use an alternate return.
index	Index that you are referencing. Set to 0 if the primary index was used in the LOCK$ call. Set to 1-17 if secondary index was used in LOCK$ call. Set to -1 if direct access. Make sure that the index setting for UPDAT$ matches its setting in the previous LOCK$ call.
file-no	(Obsolete, set to 0.)
bufsiz	(Ignored for this call.) It can be left the same as it was for LOCK$.
keysiz	(Ignored, set to 0.)

Table 5-13
UPDAT$ Flags

Flag	Function
FL$KEY	Indicates that a full primary key is present in the buffer. Use only when keys are not stored in the data records.
FL$ULK	Tells UPDAT$ to unlock the record only. All other flags are ignored and the data is not updated.
FL$USE	Required. Set this flag on. The returned array from the prior LOCK$ call is used on the UPDAT$ call. If not set on, error 30 occurs.

WARNING

You cannot increase the size of variable-length records in a call to UPDAT$. UPDAT$ only updates the length of the data that was originally declared in ADD1$. Problems occur if the bufsiz is less than but not equal to the declared size in ADD1$; MIDASPLUS updates only that portion of the record and leaves the remainder of the record unchanged. For example:

> If the data record is 12345678 (8 characters long) and you call UPDAT$ with the buffer set to NNNNNN (6 characters) and the bufsiz set to 3 words, MIDASPLUS only writes out three words and leaves the fourth word unchanged. The resulting data record is NNNNNN78.

Problems also occur if the bufsiz equals zero and the buffer is longer than the declared size; MIDASPLUS truncates the buffer to the declared size and does not produce an error message. For example:

> If the data record is 12345678 (8 characters long) and you call UPDAT$ with the buffer set to NNNNNNNN00 (10 characters) and the bufsiz set to 0 words, MIDASPLUS only writes out 4 words. The last 00 is lost; the resulting data record is NNNNNNNN.

DELET$

DELET$ removes 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. If you delete a primary key, the data record is marked for deletion and can no longer be retrieved. The secondary keys are not deleted, but since they point to a "deleted" record, they are meaningless. They will be deleted the first time that they are accessed or when MPACK is used with the file. When a secondary key is deleted, there is no effect on the data record and consequently no effect on other keys. For more information about handling deleted records, see Appendix E, CONCURRENCY ISSUES.

DELET$ deletes a record whether it is locked or unlocked. Since DELET$ both positions to and removes a record or an index subfile entry, it is not necessary to call another subroutine to first find the record or key.

DELET$ Calling Sequence

The DELET$ calling sequence is the same sequence that is shared with most of the other interface routines. MIDASPLUS ignores some of the arguments and they can be set to 0 in the call. Table 5-14 lists the arguments for DELET$.

```
Call DELET$  (funit, buffer, key, array, flags, altrn, index,
              file-no, bufsiz, keysiz)
```

Locating the Record to Delete

Either a primary or secondary key value can be used to locate the record intended for deletion. Give DELET$ the full primary or secondary key value in key and set index appropriately. You may also use a FIND$, NEXT$, or LOCK$ operation before a delete operation to find the record to be deleted. In this case, set the FL$RET flag in this prior call so that DELET$ can use the returned array. Then set FL$USE in flags in the call to DELET$. When FL$USE is set on in the call, the key and keysiz arguments are ignored.

Deleting Duplicates

When deleting duplicate key entries, you must use NEXT$ to locate the record that you want deleted. Check the record to make sure that you have the right one. Neither DELET$ nor FIND$ will work unless the record or index that you plan to delete is the oldest duplicate (that is, the first duplicate to physically appear in the index) in a

subfile. Both DELET$ and FIND$ are unacceptable for deleting duplicates because they automatically position to the oldest duplicate value for that key in the file.

Table 5-14
DELET$ Arguments

Argument	Meaning
funit	File unit on which the MIDASPLUS file is open.
buffer	Ignored.
key	Full primary or secondary key used to identify the entry to be deleted. Do not supply a value for key if you are using the array from the previous call (assumes FL$USE is set).
array	Communications array that supply array(1) as 0 or 1 in keyed-index access. Include data size in word 2 and entry number in words 3 and 4 for direct access. See DELET$ and Direct Access below.
flags	FL$USE is the only applicable flag in this call. Set FL$USE if a previous call to FIND$ or NEXT$ was made to locate the entry to be deleted. All other flag options are ignored.
altrtn	Statement number of alternate return to be used in case an error occurs on the subroutine calls. Supply 0 if you cannot use an alternate return.
index	Index that you are referencing. Indicates whether a data record (and primary index entry) or a secondary index entry should be deleted. 0 — Deletes primary index entry and the data record that it references. 1 - 17 — Deletes secondary index entry from a specified index. -1 — Deletes the primary index entry and the data record from a direct access file.

Table 5-14 (continued)
DELET$ Arguments

Argument	Meaning
file-no	(Obsolete, set to 0.)
bufsiz	(Ignored, set to 0.)
keysiz	(Ignored, set to 0.)

Deleting Secondary Index Entries

You can remove a secondary index entry without touching the data record that it references and without deleting the primary index entry associated with it. To locate the entry to be deleted, call DELET$ with FL$USE set off. Set index and key to the index number and full key value to be used in the call.

As an alternative, the secondary index to be deleted can be located with a call to FIND$ or NEXT$ with FL$RET set in flags and with index set to the appropriate secondary index subfile number. A call to either FIND$ or NEXT$ is then followed by a call to DELET$ with FL$USE set on and with the value for index unchanged. Set key to 0 since it is ignored when FL$USE is set.

Removing a Record and All Keys

In order to avoid useless entries that do not point to anything, delete the secondary index entries before the actual data record is deleted. First, delete all of the secondary index entries that reference a record, then delete the data record and its primary index key. Storing all of the keys in the data record makes this process much easier.

DELET$ and Direct Access

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

To delete a secondary index entry from a direct access file, use the same method as for an indexed file. The index number should be the index that is referenced and not -1.

FORTRAN PROGRAMMING EXAMPLE

```
C THIS PROGRAM ADDS, DELETES, AND PRINTS NAMES FROM A BANK CUSTOMER
C FILE.
C
C
C THE FORTRAN INTERFACE REQUIRES THAT THE SYSCOM>PARM.K.INS.FTN
C FILE (CONTAINS PARAMETERS USED IN FORTRAN SUBROUTINES) AND THE
C THE SYSCOM>KEYS.INS.FTN (CONTAINS KEY DECLARATIONS) FILES BE
C INSERTED AT THE BEGINNING OF EACH PROGRAM THAT USES MIDASPLUS.
C
C
$INSERT SYSCOM>PARM.K.INS.FTN
$INSERT SYSCOM>KEYS.INS.FTN
C     Declarations
      INTEGER*2 TERMINAL
      INTEGER*2 ARRAY (14)
      INTEGER*2 INDEX, FUNIT, STATUS, FLAGS, NAMELEN
      INTEGER*2 BUFSIZ, KEYSIZ, CODE, MODE
      CHARACTER*9 PKEY
      CHARACTER*1 ANSWER          /* P(PRINT), A(ADD), D(DELETE),
      CHARACTER*10  SKEY2          /* Q(QUIT)
      CHARACTER*25 SKEY1
      CHARACTER*86 BUFFER
      CHARACTER*25 KEY
      CHARACTER*16 STREET
      CHARACTER*2 STATE
      CHARACTER*15 CITY
      CHARACTER*9 ZIP

      CHARACTER*4 PATHNAME

      EQUIVALENCE (BUFFER(1:1),PKEY),
     1(BUFFER(10:10),SKEY1), (BUFFER(35:35),SKEY2),
     1(BUFFER(45:45),STREET), (BUFFER(61:61),CITY),
     1(BUFFER(76:76),STATE), (BUFFER(78:78),ZIP)

      CHARACTER*1 ANSER2          /* Y(es) or N(o)
      LOGICAL*2 SWITCH(2)
      INTEGER*2 FILE_OPEN
      MODE = K$RDWR + K$GETU
C THE BANK FILE WAS CREATED DURING CREATK
      PATHNAME = 'BANK'
      NAMELEN = 4
      TERMINAL = 1
      SWITCH(1) = .FALSE. /*SKEY1 SWITCH
      SWITCH(2) = .FALSE.        /*SKEY2
      FILE_OPEN = 0              /* FILE OPEN SWITCH
C OPEN THE BANK FILE
      CALL OPENM$ (MODE, PATHNAME, NAMELEN, FUNIT, STATUS)
C
      IF (STATUS.EQ.0) THEN
           FILE_OPEN = 1
      ELSE
```

```
          PRINT *, '**ERROR OPENING BANK FILE: ', STATUS
          GO TO 8000
        END IF
C
C QUERY USER FOR ACTION (PRINT, ADD, DELETE, OR QUIT).
C

 900  WRITE (TERMINAL, 9002) 'ENTER ACTION - P(rint), A(dd), ',
     1' D(elete) OR Q(uit):'
      READ (TERMINAL, 9001) ANSWER
      IF (ANSWER.EQ. 'P' .OR. ANSWER.EQ.'A' .OR. ANSWER.EQ.'D') THEN
          GO TO 950
      ELSE IF (ANSWER.EQ.'Q') THEN
        GO TO 8000

      ELSE
C
C THE MESSAGE "INVALID OPTION...PLEASE TRY AGAIN" APPEARS IF A
C USER ENTERS A RESPONSE OTHER THAN P, A, D, OR Q.
        PRINT *,  'INVALID OPTION...PLEASE TRY AGAIN'
        GO TO 900
      END IF
 950  CONTINUE
C
C     ADD A RECORD
C
      IF (ANSWER.EQ.'A') THEN
         GO TO 1000
      ELSE
         GO TO 2000
      END IF
C
C THE USER SELECTED THE ADD OPTION SECTION; THE PROGRAM GETS
C DATA AND ADDS IT INTO THE FILE.
C
 1000 WRITE (TERMINAL, 9001) 'ENTER 9 DIGIT CUSTOMER ID: '
      READ  (TERMINAL, 9001)  PKEY
 1010 WRITE (TERMINAL, 9001) 'ADDING CUSTOMER NAME?'
      READ  (TERMINAL, 9001) ANSER2

      IF (ANSER2.EQ.'Y') THEN
          SWITCH(1) = .TRUE.
          GO TO 1100
      ELSE IF (ANSER2.EQ.'N') THEN
          GO TO 1200      /* RECORD WILL HAVE CUST-ID# ONLY
      ELSE
C USER IS PROMPTED REPEATEDLY UNTIL THE RESPONSE IS Y OR N
          WRITE (TERMINAL, 9001)  'ANSWER Y or N'
          GO TO 1010
      END IF
C USER ENTERS CUSTOMER NAME (UP TO 25 CHARACTERS)
 1100 WRITE (TERMINAL ,9001) 'ENTER CUSTOMER NAME: '
      READ  (TERMINAL, 9001) SKEY1
```

```
C USER HAS THE OPTION OF ADDING A UNIQUE, 10 DIGIT, ALPHA
C NUMERIC ACCOUNT NUMBER.
 1120 WRITE (TERMINAL, 9001) 'ADDING ACCOUNT # ?'
      READ  (TERMINAL, 9001) ANSER2
      IF (ANSER2.EQ.'Y') THEN
          SWITCH(2) = .TRUE.
          GO TO 1150
      ELSE IF (ANSER2.EQ.'N') THEN
          GO TO 1160
      ELSE
          WRITE (TERMINAL, 9001) 'ANSWER Y or N'
          GO TO 1120
      END IF
 1150 WRITE (TERMINAL, 9001) 'ENTER 10 DIGIT ACCOUNT NUMBER: '
      READ  (TERMINAL, 9001) SKEY2
C USER ENTERS CUSTOMER ADDRESS
 1160 WRITE (TERMINAL, 9001) 'ENTER STREET ADDRESS: '
      READ  (TERMINAL, 9001) STREET
      WRITE (TERMINAL, 9001) 'ENTER CUSTOMER CITY: '
      READ  (TERMINAL, 9001) CITY
      WRITE (TERMINAL, 9001) 'ENTER CUSTOMER STATE: '
      READ  (TERMINAL, 9001) STATE
      WRITE (TERMINAL, 9001) 'ENTER ZIP CODE: '
      READ  (TERMINAL, 9001) ZIP
 1200 CONTINUE
      INDEX = 0
      KEYSIZE = 9
      FLAGS = FL$RET
C THE NEW RECORD IS ADDED TO THE FILE
      CALL ADD1$ (FUNIT, BUFFER, PKEY, ARRAY, FLAGS, $5000,
     1INDEX, 0, 0, KEYSIZE)

      IF (.NOT. SWITCH(1) ) GO TO 1210
      INDEX = 1
      KEYSIZE = 25
      FLAGS = FL$USE + FL$RET
      CALL ADD1$ (FUNIT, BUFFER, SKEY1, ARRAY, FLAGS, $5000,
     1INDEX, 0, 0, KEYSIZE)
      SWITCH (1) = .FALSE.

 1210 IF (.NOT. SWITCH(2)) GO TO 1220
      INDEX = 2
      KEYSIZ = 10
      CALL ADD1$ (FUNIT, BUFFER, SKEY2, ARRAY, FLAGS, $5000,
     1INDEX, 0 0, KEYSIZE)
      SWITCH (2) = .FALSE.
C
 1220 GO TO 900
C     ERROR MESSAGE FOR ADD: PRINTS MESSAGE, INDEX #, STATUS CODE,

 2000 CONTINUE                              /* PRINT RECORD

C'THE USER IS GIVEN THE OPTION OF LOCATING THE CUSTOMER BY THE
C ACCOUNT NUMBER.  IF THE USER ANSWERS YES, THE USER IS PROMPTED
```

```
C FOR THE CUSTOMER'S ACCOUNT NUMBER.  IF THE USER ANSWERS NO,
C THE USER IS PROMPTED FOR THE CUSTOMER'S IDENTIFICATION NUMBER.

 2010 WRITE (TERMINAL, 9001) 'LOCATE BY ACCOUNT NO.? '
      READ (TERMINAL, 9001) ANSER2
      IF (ANSER2.EQ.'N') THEN
         GO TO 2050
      ELSE IF (ANSER2.EQ.'Y') THEN
         GO TO 2040
      END IF
      PRINT *, 'PLEASE ANSWER Yes or No '
      GO TO 2010

C THE USER ENTERS THE CUSTOMER'S 10 DIGIT ACCOUNT NUMBER.

 2040 WRITE (TERMINAL, 9001) 'ENTER 10-DIGIT ACCOUNT NO: '
      READ  (TERMINAL, 9001) SKEY2
      WRITE (TERMINAL, 9001) 'READ SKEY2'
      INDEX = 2
      FLAGS = FL$RET
      CALL FIND$ (FUNIT, BUFFER,SKEY2, ARRAY, FLAGS,$5050,
     1INDEX, 0, 0, 0)
      GO TO 2100

C THE USER IS GIVEN THE OPTION OF LOCATING THE CUSTOMER BY THE
C CUSTOMER'S IDENTIFICATION NUMBER.  IF THE RESPONSE IS YES,
C THE USER IS PROMPTED FOR THE CUSTOMER'S IDENTIFICATION NUMBER.
C IF THE RESPONSE IS NO, THE MESSAGE "NO KEY SUPPLIED FOR PRINTING"
C APPEARS AND THE USER IS RETURNED TO LINE 900 - QUERY FOR ACTION.

 2050 WRITE (TERMINAL, 9001) 'LOCATE BY CUSTOMER ID? '
      READ  (TERMINAL, 9001) ANSER2
      IF (ANSER2.EQ.'N') THEN
          GO TO 2060
      ELSE IF (ANSER2.EQ.'Y') THEN
           GO TO 2070
      ELSE
           PRINT *, 'PLEASE ANSWER Yes or No'
           GO TO 2050
      END IF
 2060 PRINT *, 'NO KEY SUPPLIED FOR PRINTING'
      GO TO 900

 2070 WRITE (TERMINAL, 9001) 'ENTER CUSTOMER ID'
      READ  (TERMINAL, 9001) PKEY
      WRITE (TERMINAL, 9001) 'READ PKEY'
      INDEX = 0
C
      FLAGS = FL$RET
      CALL FIND$ (FUNIT, BUFFER,PKEY, ARRAY, FLAGS,$5050,
     1INDEX, 0, 0, 0)
C
C THE CUSTOMER NAME, ID, ADDRESS, AND ACCOUNT NUMBER ARE
C PRINTED.
```

```
2100  PRINT *, 'ACCOUNT# =  ', SKEY2
      WRITE (TERMINAL, 9002) 'CUSTOMER NAME:     ', SKEY1
      WRITE (TERMINAL, 9002) 'CUSTOMER ID#:      ', PKEY
      WRITE (TERMINAL, 9002) 'CUSTOMER ADDRESS: ', STREET
      WRITE (TERMINAL, 9002) '                   ', CITY
      WRITE (TERMINAL, 9004) '                   ', STATE, ZIP
C     IF PRINT REQUESTED, THEN DONE, GO BACK FOR NEXT REQUEST
 2110 CONTINUE
      IF (ANSWER.EQ.'P') THEN
          GO TO 900
      END IF
C
C THE USER IS ASKED WHETHER THE DISPLAYED RECORD SHOULD BE DELETED.
C IF THE RESPONSE IS YES, THE RECORD IS DELETED.  IF THE RESPONSE
C IS NO, USER IS RETURNED TO LINE 900 - QUERY FOR ACTION.
C
C DELETE SECTION
 2500 WRITE (TERMINAL, 9001) 'OKAY TO DELETE THIS RECORD?'
      READ  (TERMINAL, 9001) ANSER2
      IF (ANSER2.EQ.'Y') THEN
         GO TO 3000
      ELSE IF (ANSER2.EQ.'N') THEN
         GO TO 900
      END IF
      PRINT *, 'PLEASE ANSWER Yes or No'
      GO TO 2500

 3000 ARRAY (1) = 0
      INDEX = 2
      CALL DELET$ (FUNIT, BUFFER, SKEY2, ARRAY, FLAGS,
     1$5100, INDEX, 0, 0, 0)
      INDEX = 1
      CALL DELET$ (FUNIT, BUFFER, SKEY1, ARRAY, FLAGS,
     1$5100, INDEX, 0, 0, 0)
      INDEX = 0
      CALL DELET$ (FUNIT, BUFFER, PKEY, ARRAY, FLAGS,
     1$5100, INDEX, 0, 0, 0)

      GO TO 900
 5000 WRITE (TERMINAL, 9001) 'ERROR ON ADD, KEY = '
      PRINT *, PKEY
      GO TO 900

 5050 WRITE (TERMINAL, 9001) 'ERROR ON FIND, KEY = '
      PRINT *,  PKEY
      GO TO 900

 5100 WRITE (TERMINAL, 9001) 'ERROR ON DELETE'
      PRINT *,  PKEY
      GO TO 900

 8000 IF (FILE_OPEN .EQ. 1) THEN
C USER REQUESTED EXIT.
```

```
            PRINT *, 'NOW CLOSING FILE'
            CALL CLOSM$ (FUNIT, STATUS)
            IF (STATUS.NE.0) THEN
            PRINT *, '**ERROR CLOSING BANK FILE: ', STATUS
            ELSE
            FILE_OPEN = 0
            END IF
        ELSE
            PRINT *, 'FILE NOT OPEN'
        END IF
C
 8010 PRINT *, 'PROGRAM COMPLETED.'
        CALL EXIT
C END OF EXECUTABLE CODE.
C
C FORMAT STATEMENTS
 9001 FORMAT (A)
 9002 FORMAT ((A),(A))
 9003 FORMAT ((A),(A),(A))
 9004 FORMAT ((A),(A),(A),(A))
        CONTINUE
        END

OK, F77 CUST -INTS
[F77 Rev. 19.4]
0000 ERRORS [<.MAIN.> F77 Rev. 19.4]
OK, BIND
[BIND rev 19.4.1]
: LOAD CUST
: LI MPLUSLB
: LI
BIND COMPLETE
: FILE
OK, RESUME CUST
ENTER ACTION - P(rint), A(dd),  D(elete) OR Q(uit):
A
ENTER 9 DIGIT CUSTOMER ID:
28276503889
ADDING CUSTOMER NAME?
Y
ENTER CUSTOMER NAME:
HARPER, ANNE
ADDING ACCOUNT # ?
Y
ENTER 10 DIGIT ACCOUNT NUMBER:
CHK4123891
ENTER STREET ADDRESS:
12 WASHINGTON ST
ENTER CUSTOMER CITY:
NEWTON
ENTER CUSTOMER STATE:
MA
ENTER ZIP CODE:
02159
```

```
ENTER ACTION - P(rint), A(dd),  D(elete) OR Q(uit):
A
ENTER 9 DIGIT CUSTOMER ID:
32023677386
ADDING CUSTOMER NAME?
Y
ENTER CUSTOMER NAME:
CORRADO, THOMAS
ADDING ACCOUNT # ?
Y
ENTER 10 DIGIT ACCOUNT NUMBER:
SAV1273565
ENTER STREET ADDRESS:
42 MAPLE AVE
ENTER CUSTOMER CITY:
ARLINGTON
ENTER CUSTOMER STATE:
MA
ENTER ZIP CODE:
02174
ENTER ACTION - P(rint), A(dd),  D(elete) OR Q(uit):
P
LOCATE BY ACCOUNT NO.?
Y
ENTER 10-DIGIT ACCOUNT NO:
CHK4123891
READ SKEY2
 ACCOUNT# =  CHK4123891
CUSTOMER NAME:     HARPER, ANNE
CUSTOMER ID#:      282765038
CUSTOMER ADDRESS: 12 WASHINGTON ST
                   NEWTON
                   MA02159
ENTER ACTION - P(rint), A(dd),  D(elete) OR Q(uit):
P
LOCATE BY ACCOUNT NO.?
N
LOCATE BY CUSTOMER ID?
Y
ENTER CUSTOMER ID
32023677386
READ PKEY
 ACCOUNT# =  SAV1273565
CUSTOMER NAME:     CORRADO, THOMAS
CUSTOMER ID#:      320236773
CUSTOMER ADDRESS: 42 MAPLE AVE
                   ARLINGTON
                   MA02174
ENTER ACTION - P(rint), A(dd),  D(elete) OR Q(uit):
D
LOCATE BY ACCOUNT NO.?
Y
ENTER 10-DIGIT ACCOUNT NO:
SAV1273565
```

```
READ SKEY2
 ACCOUNT# =  SAV1273565
CUSTOMER NAME:     CORRADO, THOMAS
CUSTOMER ID#:      320236773
CUSTOMER ADDRESS: 42 MAPLE AVE
                  ARLINGTON
                  MA02174
OKAY TO DELETE THIS RECORD?
Y
ENTER ACTION - P(rint), A(dd),  D(elete) OR Q(uit):
Q
 NOW CLOSING FILE
 PROGRAM COMPLETED.
OK,
```

6

The COBOL Interface

This chapter discusses the COBOL interface to MIDASPLUS. The COBOL interface to MIDASPLUS uses the Prime CBL compiler and is based on the standard COBOL I/O statements for INDEXED and RELATIVE files. Keyed-index access MIDASPLUS files are called INDEXED files in COBOL and direct access MIDASPLUS files are known as RELATIVE files. You can access MIDASPLUS files through the COBOL interface just as if they were any other standard COBOL INDEXED or RELATIVE file.

You must create a template with CREATK for both INDEXED and RELATIVE files. While COBOL can access an existing file, it cannot create a MIDASPLUS file from the program level. Using CREATK, a template has been created for the sample MIDASPLUS file referred to in this chapter.

This chapter explains how to access both INDEXED files and RELATIVE files from a COBOL program. It explains the syntax of COBOL statements used to read, write, and update records in a MIDASPLUS file. The chapter also describes how to define the file's characteristics in the different parts of a COBOL program. See the Prime manual that documents COBOL for detailed information on COBOL's syntax and concepts.

LANGUAGE DEPENDENCIES

Certain rules exist about keys and record sizes in MIDASPLUS files accessed by COBOL applications.

The rules about keys are:

- Up to 17 secondary keys are supported per INDEXED file.
- The primary key and any secondary keys must be included in the data record.
- While the primary key can be anywhere in the data record, secondary keys cannot be embedded in the primary key. (If you change any of the secondary key values, you will affect the primary key field which cannot be changed.)
- Secondary key index subfiles may not contain any secondary data.
- The maximum ASCII key size is 64 characters.
- The maximum bit string key size is 32 characters.

The rules about record size are:

- If a MIDASPLUS file has fixed-length records, the record size indicated in the COBOL program must match the data size defined for the file during CREATK.
- Variable-length records are supported for INDEXED files; however, COBOL may require you to set minimum and maximum record sizes for the file, sizes that match those in the program. For details on setting record size limits, see the section VARIABLE-LENGTH RECORDS AND SPACE USAGE, in Chapter 2.

Note

Restrictions for RELATIVE files are covered in the section DIRECT ACCESS FILES IN COBOL in this chapter.

Compiling and Loading a COBOL Program

The following is a compile and load sequence that shows all of the libraries that must be loaded to run a program:

```
OK, cbl program

[CBL rev 19.4]
OK, bind
[BIND rev 19.4]
: load program
: li cbllib
: li
BIND COMPLETE
: file
OK, resume program
```

Substitute the appropriate program name for the program parameter shown in the above sequence.

SUMMARY OF COBOL STATEMENTS

Table 6-1 summarizes the COBOL statements needed to process MIDASPLUS files.

Table 6-1
Summary of COBOL Statements

Statement	Function
OPEN	Opens the MIDASPLUS file and establishes the access mode. Execute this statement before any other statement that references the file.
CLOSE	Closes the MIDASPLUS file and causes the file unit on which the file is opened to be released.
USE AFTER	Defines a procedure that will be executed if an INVALID KEY clause or an AT END clause is not specified.
START	Moves the file pointer to a specific record in the file and establishes the file position in a MIDASPLUS file opened for or DYNAMIC access.
WRITE	Adds records to a file.
REWRITE	Replaces the current record with a new text string and destroys the original. REWRITE does not establish or change file position.
DELETE	Removes the data record and its primary index entry.
READ	Retrieves records from a file.

This chapter explains these statements in detail.

DEFINING AN INDEXED MIDASPLUS FILE

The rules for defining an INDEXED file in the File Control Section and in the Data Division of a program are discussed below.

The primary key in an INDEXED file is called the RECORD KEY. The secondary keys are called ALTERNATE RECORD KEYS. Prime's COBOL supports the use of up to 17 secondary keys in INDEXED files.

FILE-CONTROL Requirements

The FILE-CONTROL paragraph contains

- The internal names of the files to be accessed
- The names of the devices on which they are to be opened (PFMS)
- The access mode specifications
- The names of the primary key (RECORD KEY) (one for each file)
- The names of any secondary keys (ALTERNATE RECORD KEY) present in each file
- A file status, which if present, is used to monitor the success or failure of each operation

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 paragraph rules, refer to the COBOL 74 Reference Guide. The following is a summary of the important rules.

The SELECT Clause: SELECT defines the name of the MIDASPLUS file and tells the compiler to assign it some available file unit. Always assign the file to PFMS (disk).

The ORGANIZATION Clause: ORGANIZATION tells the compiler that the file to be opened is a keyed-index MIDASPLUS file.

The ACCESS MODE Clause: This clause is optional; the default mode is SEQUENTIAL. If SEQUENTIAL is specified, or if the clause is omitted, you must perform all reads and writes sequentially. No random operations are allowed. Add records in primary key order and retrieve them in key order in SEQUENTIAL access mode.

If you choose the RANDOM access mode, write and retrieve the records in a random fashion, based on a supplied key value. Sequential reads and writes are not permitted.

The DYNAMIC access mode lets you read and write sequentially or randomly. You can switch back and forth between the two.

The RECORD KEY Clause: RECORD KEY defines the key-name associated with the primary key for the MIDASPLUS file. Define the parameter key-name-1 in the Record Description entry associated with this file's FD entry. The parameter key-name-1 can be anywhere in this description.

The following rules apply to RECORD KEY definition:

- Do not specify a primary key with an OCCURS clause.
- The length of the primary key cannot exceed 64 characters if it is an ASCII key or 32 characters if it is a bit string.
- The primary key must be the same length and type as that defined during template creation.
- The primary key cannot have a P character in its PICTURE clause.
- The primary key cannot be defined as numeric with a separate sign.
- Do not embed any secondary keys within the primary key. (The primary key value cannot be changed.)
- The primary key cannot be defined in the WORKING-STORAGE section.

If you are not sure of the key length or type when defining the keys in the File-Control paragraph, use the PRINT option of CREATK to get a summary of each index description.

The ALTERNATE RECORD KEY Clause: ALTERNATE RECORD KEY designates a field within each record as a secondary key. As stated earlier, you may specify a MIDASPLUS file's secondary keys during template creation. In COBOL always define the keys in the order that they were created during CREATK, (that is, index 1, index 2 ...). The ALTERNATE RECORD clause tells COBOL about the order and length of each field that you designated as a key for this MIDASPLUS file. Use a separate ALTERNATE RECORD KEY clause for each secondary key you defined in the template. Use the WITH DUPLICATES modifier, a documentation feature, only for those keys that were given duplicate status during CREATK. You cannot change the duplicate status of an index at the program level. (Only CREATK can change this status.)

Secondary keys are bound by the same size and type restrictions as the primary key and cannot have P characters in their PICTURE clauses. Do not define secondary keys in the program's WORKING-STORAGE section. Remember that secondary keys apply to INDEXED files only.

The FILE STATUS Clause: Names a two-byte unsigned field declared in WORKING-STORAGE, called status-code. COBOL's I/O uses this field to indicate the execution status of each program statement that references the file. Each time an I/O statement 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 Appendix B, ERROR MESSAGES.

DATA DIVISION Requirements

The FILE SECTION of the DATA DIVISION describes the record structure of each file mentioned in the FILE-CONTROL paragraph. The WORKING-STORAGE section may describe data items which are not part of files but which are used to handle data written to and read from these files during program execution.

The FILE SECTION of the DATA DIVISION consists of the following:

- One or more file description entries called FDs.
- One file-id value, which defines the actual name of the MIDASPLUS file.
- An FD may have one or more record descriptions. If an FD has more than one record description, the key must be in the same relative position in each record.

The general format of a File Description entry in CBL is:

```
FD filename   EXTERNAL

LABEL { RECORD IS  }
      { 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 }

[OWNER-IS literal-1]

record description-entry...
```

Note the following for the File Section clauses:

- The name that the program uses to refer to this MIDASPLUS file must follow the FD clause. The LABEL RECORD IS STANDARD clause is not required for disk files with the CBL compiler, but is required for the COBOL compiler.

- If you use the optional RECORD CONTAINS clause, make sure that the number of specified characters matches the data record size specified during template creation. The maximum record size is 32767 characters.

- If you use the DATA RECORD clause, it must name the record description(s) that follow the FD entry. If more than one record description is defined per file, give a separate description of each one. Begin each new record description with an 01 level number. Multiple record descriptions imply that a file has more than one record description, but all share the same buffer area. Specify the key fields in the same relative position within each record description.

- The VALUE OF FILE-ID clause is used to tie an internal filename to the actual name of the MIDASPLUS file as it appears on disk. If this clause is omitted, the internal filename is the default.

The record description defines all of 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 74 Reference Guide.

The OPEN Statement

The OPEN statement opens the MIDASPLUS file and establishes the access mode. Execute it before any other statement that references the file. You can open more than one file 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:

```
OPEN { INPUT  }
     { I-O    }  filename-1 [,filename-2, ...]
     { OUTPUT }
```

The filename is the internal name as specified in the SELECT clause. You may apply the INPUT, OUTPUT, I-O, or open modes to this file.

If MIDASPLUS cannot locate the named file based on the actual file name, the program will abort at runtime.

You can use this statement to open more than one file as shown in this example:

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

Open Modes: The open mode determines the operations you can perform on a file, as follows:

- In INPUT mode, only READ statements can access a file.
- In OUTPUT mode, only WRITE statements can write to a file.
- In I-O mode, a file can be both read and written to and records can be updated and deleted. Records are automatically locked when read in I-O mode, whereas they are not locked in INPUT mode. (See Record Locking later in this chapter.)

Table 6-2 shows what statements can be used in each access mode.

Table 6-2
Statements Permitted in Each Access Path

File Access Mode	Statement	Open Mode		
		INPUT	OUTPUT	I-O
SEQUENTIAL	READ	●		● *
	WRITE		●	● **
	REWRITE			●
	START	●		●
	DELETE			●
RANDOM	READ	●		● *
	WRITE		●	●
	REWRITE			●
	START			
	DELETE			●
DYNAMIC	READ	●		● *
	WRITE		●	●
	REWRITE			●
	START	●		●
	DELETE			●

*Records are locked.
**Indexed files only.

The CLOSE Statement

The CLOSE statement is the reverse of the OPEN statement. It causes the file unit on which the file is opened to be released. The form is:

```
CLOSE filename-1 [, filename-2, ...]
```

filename is the name of the file specified in the SELECT and FD clauses. You can open and close a file more than once in the same program. Attempts to open a file which has not been closed, however, will result in a runtime abort.

ERROR HANDLING

One of the following three clauses of handling runtime errors must be specified for each I/O verb.

- The AT END clause
- The INVALID KEY clause
- The USE AFTER ERROR statement

A brief explanation of how these error handlers work follows. Refer to the COBOL 74 Reference Guide for complete details on error handlers.

The AT END Clause

The AT END clause, used only in a sequential READ statement (access mode is DYNAMIC or SEQUENTIAL), prevents program failure when an end-of-file condition is met during the read. The format is:

```
READ filename AT END imperative-statement
```

An illustration of the imperative-statement might be the use of a PERFORM statement that transfers control to another procedure that performs some further useful action or just closes the file.

The INVALID KEY Clause

The INVALID KEY clause identifies and traps MIDASPLUS errors. Use it in START, READ, WRITE, READ, REWRITE, and DELETE statements to protect your program from key errors. Without an INVALID KEY clause or a USE AFTER statement, the program aborts when an I/O operation is unsuccessful.

Three exceptions to the above usages of INVALID KEY are:

- SEQUENTIAL ACCESS READ statements (use AT END statement and/or USE AFTER statement)
- SEQUENTIAL or DYNAMIC ACCESS READ NEXT statements (use AT END statement and/or USE AFTER statement)
- SEQUENTIAL ACCESS DELETE statements (use optional USE AFTER statement)

The format of the INVALID KEY clause is:

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

The word KEY is optional. When this clause is executed, examine the status code variable specified in the FILE STATUS clause of the FILE CONTROL section to determine the cause of the error. See Appendix B, ERROR MESSAGES, for a list of codes.

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 MIDASPLUS's MFILE file occurs during this read, the READ-ERROR procedure is performed. This procedure might test status-code for the various errors, and perform an appropriate operation to recover from that error.

The USE AFTER Statement

Place the USE AFTER statement under the DECLARATIVES section of the program, immediately following a section header (and a period and a space). This statement defines a procedure that is executed in one of three cases: if an INVALID KEY clause is omitted; if an AT END clause is omitted; or if an AT END clause is supplied, but a non-end-of-file error occurs.

The format of USE AFTER is:

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

Use the INPUT, OUTPUT, I-O, and filename parameters to indicate when that particular procedure should be executed. When filename is specified, this procedure only handles errors occurring while processing that file. USE AFTER is never executed. It identifies the conditions under which the procedure it introduces should be executed. The terms EXCEPTION and ERROR have the same meaning.

```
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 that follows the invoking statement. The following is an example of a 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 one procedure for INPUT errors, another one for OUTPUT errors, and another one for I-O errors.

FILE POSITION

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 that change the current record location are OPEN, START, READ and DELETE. DELETE leaves the current record position undefined after a record is deleted. After a DELETE, you can do a sequential or keyed READ to reestablish the current position.

File Positioning

File positioning is done relative to a primary or secondary index subfile. You see file positioning in terms of which record is returned at any given point. The file position is based on the key that you supply 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. MIDASPLUS uses the order of entries in the primary index subfile as a basis for finding and returning data subfile records.

Likewise, if a secondary key value is used in a START or a keyed read, the secondary index subfile becomes the basis for file processing. A subsequent sequential read returns the next data subfile record referenced by the next sequential entry in the secondary index subfile references. MIDASPLUS always adds entries to the index subfiles in sorted order. MIDASPLUS inserts things where you would logically expect them to be inserted.

Record Locking

Record locking applies to files opened for I-O only. The READ statement always locks the record to which it positions. This action, which happens only in files opened for I-O, protects users from conflicting updates and makes sure that you will update or delete the current record. Generally, locking protects the record from harm by any other user as long as the record remains locked. (The START and Locked Records section of this chapter describes an exception to this issue.) 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.

Accessing a Locked Record: If your program tries to access a record that someone else has already locked, a MIDASPLUS error occurs. A file status code of 90 is returned. To avoid abnormal program termination, make sure that your program handles all of the file status conditions listed in Appendix B, ERROR MESSAGES.

The START Statement

The START statement moves the file pointer to a specific record in the file. This establishes the file position in a MIDASPLUS file opened for SEQUENTIAL or DYNAMIC access. Do not use START in a file opened for RANDOM access.

To position a MIDASPLUS file to a particular record, START uses a specific key value or a conditional expression based on a key value. Follow these steps to position the file:

1. Use a MOVE statement to assign an 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 one of the following:

   - The first record containing that key value
   - The first record with a key value greater than the value assigned to that key
   - 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].
```

Note

The symbols >, NOT <, or = may also be used.

key-name is the name of a file key and contains the value that the MOVE operation previously assigned. The START statement uses the assigned value in key-name for comparison. Include the INVALID KEY clause unless the DECLARATIVES have provided a USE AFTER procedure.

Some important points to note about START are:

- START only positions the file pointer; it does not return the record (as in a READ).
- Only use START with files opened for SEQUENTIAL or DYNAMIC access.

- If you are using the EQUAL TO option and the key value specified in the previous MOVE does not exist, the program terminates abnormally unless an error-handling mechanism is included in the program. A file status code of 23 is returned.

- You can use both primary and secondary key values to position the file. Assign a value to either the RECORD KEY (primary key) or to an ALTERNATE RECORD KEY (a secondary key) before the START statement. If you use a secondary key, include the KEY IS key-name clause 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 a secondary key that does not allow duplicates, the EQUAL TO option positions to the record in which the key field value is the same as the value assigned to key-name.

- If key-name is a secondary key that allows duplicates, the EQUAL TO option positions to the first record with the indicated key value.

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

- If a record is not found in the file that satisfies the comparison specified in the START statement, an INVALID KEY condition exists and the position of the current record pointer is undefined.

START and Locked Records: If you attempt a START operation on a record that another user has already locked, MIDASPLUS returns a status code of 0 (successful START). However, if you attempt to READ that record, you receive a status code of 90, indicating that the record is locked. START and READ do not unlock the record for the other user.

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.
```

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

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

Positioning on Partial Keys: You can use partial keys to position the file 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 you fully initialize the key value before the START statement is executed. This applies to both primary and secondary keys. For example, using the BANK file, if you wanted to find all of 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 BANK
    MOVE 'F' TO CUST-NAME.
    START BANK KEY IS NOT LESS THAN CUST-NAME
    INVALID KEY GO TO KEY-ERR.
```

READING A FILE

File reads can be either sequential or keyed.

Sequential reads mean reading one record after the other in primary key or secondary key order depending on the index to which the file is positioned. In this type of read, you do not supply a key value except to tell MIDASPLUS where in the index file to start reading sequentially.

Keyed reads are also called random reads because it is possible to specify a new key value for searching and jumping anywhere from the current file position.

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 you to switch from one type of read to another, allowing you to 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.

Note

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

Sequential Reads

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

In SEQUENTIAL Access Mode: You use the primary or secondary key to read the file sequentially. You cannot read records 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 following the READ clause, moves the record into the read-var. If omitted, the record value is returned in the buffer associated with the file in the FD. Include the AT END clause in each READ statement, unless an applicable USE AFTER procedure is specified for this file under the DECLARATIVES. The NEXT RECORD clause is implied for each READ statement although not shown in this format. 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. They are locked, however, if the file is opened for I-O. The current record remains locked until another I/O operation is performed, yielding a new current record.

In DYNAMIC Access Mode: To read sequentially, use the NEXT clause to read a file sequentially by primary or secondary key. A START or a keyed read can establish the key on which the READ is done. 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. Specify this clause if there is not an applicable USE AFTER procedure under the DECLARATIVES.

Keyed Reads

To perform keyed (random) reads, specify 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 file-name RECORD [INTO data-name-1]
 [KEY IS data-name-2]
 [INVALID KEY Imperative-statement]
```

A keyed read eliminates the need for a START operation. If the record for which a key value has been supplied cannot be found, the INVALID KEY clause is activated. A file status code of 23 is returned. STARTs are illegal in RANDOM access mode, which allows only keyed reads. 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 points at the time the READ operation is encountered).

Partial Key Access

Partial key access is possible only if you use the MOVE and START statements (not available in RANDOM access). You cannot use partial values in READ operations. The MOVE and START operations, however, provide a good method of searching for values less than or greater than a particular value. The value may represent a full or partial key value. Partial key value means a prefix of a full key value. For example, if a value is BOSTON, legal prefixes include B, BO, BOS, and so forth.

Changing Search Indexes

The KEY IS clause allows you to switch from one index subfile to another without using a START. Put the key value that you want to search for into the proper key-name variable; then use that key-name in the KEY IS clause. This establishes key-name as the new key of reference and automatically puts you into the corresponding index subfile. If the record for which a key value has been supplied cannot be found, the INVALID KEY clause is activated. A file status code of 23 is returned.

Reading Duplicates

For secondary keys that allow duplicates, you can retrieve all of 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, for example:

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

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

```
START filename KEY IS 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 10). This condition exists when there are no more entries in the file.

4. Compare the value just read with the value sought. Verify that it is a valid duplicate.

ADDING RECORDS

You can add records to a MIDASPLUS file when it is opened for OUTPUT or I-O. The WRITE statement takes information that you supplied and adds it to the MIDASPLUS file.

Regardless of the order in which the records are presented, MIDASPLUS inserts all primary key entries into the primary index subfile in ascending key sequence (low values first). However, it always adds the data records to the bottom of the data subfile.

Like primary key entries, MIDASPLUS adds secondary key entries to secondary index subfiles in sorted order. When MIDASPLUS first tries to add a duplicate entry (for a secondary index that allows duplicates), it sets a flag in the original entry. The flag indicates that there is more than one occurrence of this particular entry value in the index subfile. It adds duplicates sequentially thereafter, following the last matching key.

Using the WRITE Statement

Supply a unique key value for the primary key of each record added to an INDEXED SEQUENTIAL file. 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 before the execution of the WRITE statement. The WRITE statement format is:

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

When using this statement:

- Make sure from-area and record-name do not reference the same memory location.
- The record from the from-area is moved to the record-name area prior to the WRITE and is truncated or blank filled.
- Supply a unique value for the primary key before the execution of each WRITE.
- Use the INVALID KEY clause to trap duplicate primary or secondary key errors. This is required unless you specify a USE AFTER procedure for this file in the DECLARATIVES.

Although you get better performance with sorted input, you can give unsorted input to the program as well.

UPDATING RECORDS (REWRITE)

The REWRITE statement replaces the current record with a new text string and destroys the original. REWRITE does not establish or change file position. You can change any field with the exception of the primary key field. Since a record must be locked in order to be updated, you can only update the current record. In SEQUENTIAL access mode, READ the record to indicate which one will be rewritten. In RANDOM mode, position to the record with a keyed read. Either of these methods is acceptable in DYNAMIC mode. If the record to be updated is not read before a REWRITE, a status code of 91 (unlocked record) is returned. In addition, in all access modes, the file must be open for I-O.

The REWRITE Format

The REWRITE statement format is the same for all access modes. If there is no USE AFTER procedure specified for this file under the DECLARATIVES, include the INVALID KEY clause in all REWRITE statements.

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

If the FROM option is used, make sure the RECORD KEY value is the same as the key used in the previous READ. This option allows you to write the new record 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) that contains the just-read data and then write it back to the file.

For variable-length records, record-name must be the same size as the record being replaced.

DELETING RECORDS

COBOL's DELETE statement removes the primary index and marks the data record for deletion. The space that secondary index entries and data entries occupy is not reclaimed until the MPACK utility is run on this file. The file must be opened for I-O in order to delete entries from it.

The DELETE Format

The DELETE format is:

```
DELETE filename RECORD
[INVALID KEY imperative-stmt].
```

filename is the name assigned to the MIDASPLUS file in the SELECT clause and FD clause. When the file is opened for RANDOM or DYNAMIC access and there is no USE AFTER procedure specified for this file, include the INVALID KEY clause in the DELETE statement. Do not include the INVALID KEY clause in DELETE statements used on files opened for SEQUENTIAL access.

A few reminders are:

- In SEQUENTIAL access mode, the record must first be read in order to be deleted. This is necessary because a DELETE operation in SEQUENTIAL access mode does not perform a position operation; the READ does the positioning.

- In SEQUENTIAL access mode, do not change the value in the RECORD KEY (primary key) between the READ and DELETE statements. DELETE can only operate on the current record. DELETE uses the primary key value, used in the READ, to check that it is the same key (or value) as the current record's primary key value.

- If the record for which a key value has been supplied cannot be found in the DYNAMIC and RANDOM access modes, the INVALID KEY clause is activated. A file status code of 23 is returned.

- A DELETE operation leaves the current record pointer undefined. A READ NEXT or a keyed read operation immediately after a DELETE will be successful, unless you deleted the last record in the file.

- You cannot perform two deletes in a row without an intervening READ in SEQUENTIAL access mode. If you supply a new primary key value, you can perform two deletes in a row in RANDOM or DYNAMIC modes.

INDEXED PROGRAMMING EXAMPLE

The following indexed program adds names to the BANK file that was created in Chapter 2.

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID.       NEW.
       INSTALLATION.     PRIME COMPUTER, INC.
       DATE-WRITTEN.     02/06/85.
       DATE-COMPILED.
       REMARKS.  THIS PROGRAM IS USED TO ADD NEW NAMES TO A BANK
                 CUSTOMER FILE.

       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       SOURCE-COMPUTER.  PRIME-750.
       OBJECT-COMPUTER.  PRIME-750.

       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT BANK-FILE ASSIGN TO PFMS
                  ORGANIZATION IS INDEXED
                  ACCESS IS DYNAMIC
                  RECORD KEY IS CUSTOMER-ID
                  ALTERNATE RECORD KEY IS CUST-NAME WITH DUPLICATES
                  ALTERNATE RECORD KEY IS ACCT-NUM
                  FILE STATUS IS FILE-STAT.

       DATA DIVISION.
       FILE SECTION.
       FD  BANK-FILE
           VALUE OF FILE-ID IS 'BANK'.
       01  BANK-REC.
           05  CUSTOMER-ID  PIC X(9).
           05  CUST-NAME    PIC X(25).
           05  ACCT-NUM     PIC X(10).
           05  CUST-ADDRESS.
               10  STREET   PIC X(16).
               10  CITY     PIC X(15).
               10  STATE    PIC XX.
               10  ZIP      PIC X(9).

       WORKING-STORAGE SECTION.

       01  COMMAND          PIC X VALUE SPACES.
           88  ADD-COMMAND      VALUES 'A', 'a'.
           88  PRINT-COMMAND    VALUES 'P', 'p'.
           88  QUIT-ENTERED     VALUES 'Q', 'q'.

       01  WORK-REC.
           05  WS-CUSTOMER-ID  PIC X(9).
           05  WS-CUST-NAME    PIC X(25).
           05  WS-ACCT-NUM     PIC X(10).
```

```
    05  WS-CUST-ADDRESS.
        10  WS-STREET    PIC X(16).
        10  WS-CITY      PIC X(15).
        10  WS-STATE     PIC XX.
        10  WS-ZIP       PIC X(9).

01  FILE-STAT  PIC 99.
01  SEARCH-TYPE  PIC X.

PROCEDURE DIVISION.
MAINLINE-CONTROL.
     OPEN I-O BANK-FILE.
     PERFORM GET-COMMAND THRU GC-EXIT
             UNTIL QUIT-ENTERED.
     CLOSE BANK-FILE.
     STOP RUN.

GET-COMMAND.
     MOVE SPACES TO COMMAND.
     DISPLAY ' '.
     DISPLAY 'Enter Command (A-add P-print Q-quit): '
         WITH NO ADVANCING.
     ACCEPT COMMAND.
     IF QUIT-ENTERED
        GO TO GC-EXIT
     ELSE IF ADD-COMMAND
        PERFORM ADD-ROUTINE THRU AR-EXIT
     ELSE IF PRINT-COMMAND
        PERFORM PRINT-ROUTINE THRU PR-EXIT
     ELSE
        DISPLAY '** ERROR = Invalid Command, try again'.
GC-EXIT.
    EXIT.

ADD-ROUTINE.
     MOVE SPACES TO CUSTOMER-ID.
     DISPLAY 'Enter CUSTOMER-ID (9 digits): '
              WITH NO ADVANCING.
     ACCEPT CUSTOMER-ID.
*    check to make sure key is not in file
     READ BANK-FILE
          INVALID KEY
             NEXT SENTENCE.
     IF FILE-STAT = 00
          DISPLAY '** ERROR: CUSTOMER-ID already in file: ',
                  CUSTOMER-ID
          GO TO AR-EXIT.
*  ok, key is not in file, get information and write record
     PERFORM GET-NEW-REC.
     WRITE BANK-REC
       INVALID KEY
         DISPLAY 'WRITE-ERROR: ' FILE-STAT.
     IF FILE-STAT = 00
         DISPLAY CUSTOMER-ID ' has been added'.
```

```
AR-EXIT.
     EXIT.

GET-NEW-REC.
      MOVE SPACES TO CUST-NAME, ACCT-NUM, STREET,
                     CITY, STATE, ZIP.
      DISPLAY 'ENTER CUSTOMER NAME: ' WITH NO ADVANCING.
      ACCEPT CUST-NAME.
      DISPLAY 'ENTER ACCOUNT NUMBER (10 digits): '
          WITH NO ADVANCING.
      ACCEPT ACCT-NUM.
      DISPLAY 'ENTER STREET ADDRESS: ' WITH NO ADVANCING.
      ACCEPT STREET.
      DISPLAY 'ENTER CITY: ' WITH NO ADVANCING.
      ACCEPT CITY.
      DISPLAY 'ENTER STATE: ' WITH NO ADVANCING.
      ACCEPT STATE.
      DISPLAY 'ENTER ZIP: ' WITH NO ADVANCING.
      ACCEPT ZIP.

PRINT-ROUTINE.

     DISPLAY
     "ENTER 'A' FOR ACCOUNT NUMBER OR 'C' FOR CUSTOMER-ID: "
         WITH NO ADVANCING.
     ACCEPT SEARCH-TYPE.
     MOVE SPACES TO BANK-REC.
     IF SEARCH-TYPE = 'C'
        MOVE SPACES TO CUSTOMER-ID
        DISPLAY 'Enter CUSTOMER-ID (9 digits): '
           WITH NO ADVANCING
        ACCEPT CUSTOMER-ID
        READ BANK-FILE
          INVALID KEY
             DISPLAY '** ERROR: NO SUCH CUSTOMER'
             GO TO PR-EXIT
     ELSE
       IF SEARCH-TYPE = 'A'
          MOVE SPACES TO ACCT-NUM
          DISPLAY 'Enter ACCT-NUM (10 digits): '
              WITH NO ADVANCING
          ACCEPT ACCT-NUM
          READ BANK-FILE KEY IS ACCT-NUM
          INVALID KEY
             DISPLAY '** ERROR: NO SUCH ACCOUNT NUMBER'
             GO TO PR-EXIT
     ELSE
       DISPLAY '** ERROR: INVALID RESPONSE, TRY AGAIN'
       GO TO PRINT-ROUTINE.

     DISPLAY 'CUSTOMER ID: ', CUSTOMER-ID.
     DISPLAY 'NAME       : ', CUST-NAME.
     DISPLAY 'ADDRESS    : ', STREET.
```

```
           DISPLAY '                ', CITY, ' ' STATE, ' ' ZIP.

       PR-EXIT.
            EXIT.
OK, CBL NEW

[CBL rev 19.4]
OK, BIND
[BIND rev 19.4.1]
: LOAD NEW
: LI CBLLIB
: LI
BIND COMPLETE
: FILE
OK, RESUME NEW

Enter Command (A-add P-print Q-quit): A
Enter CUSTOMER-ID (9 digits): 189264289
ENTER CUSTOMER NAME: MURRAY, PAUL
ENTER ACCOUNT NUMBER (10 digits): MC28374641
ENTER STREET ADDRESS: 23 ORCHARD RD
ENTER CITY: MANCHESTER
ENTER STATE: NH
ENTER ZIP: 03102
189264289 has been added

Enter Command (A-add P-print Q-quit): A
Enter CUSTOMER-ID (9 digits): 28276503889
ENTER CUSTOMER NAME: HARPER, ANNE
ENTER ACCOUNT NUMBER (10 digits): CHK4123891
ENTER STREET ADDRESS: 12 WASHINGTON ST
ENTER CITY: NEWTON
ENTER STATE: MA
ENTER ZIP: 02159
282765038 has been added

Enter Command (A-add P-print Q-quit): P
ENTER A FOR ACCOUNT NUMBER OR C FOR CUSTOMER-ID:     C
Enter CUSTOMER-ID (9 digits): 189264289
CUSTOMER ID: 189264289
NAME       : MURRAY, PAUL
ADDRESS    : 23 ORCHARD RD
             MANCHESTER      NH 03102

Enter Command (A-add P-print Q-quit): P
ENTER A FOR ACCOUNT NUMBER OR C FOR CUSTOMER-ID:     A
Enter ACCT-NUM (10 digits): CHK4123891
CUSTOMER ID: 282765038
NAME       : HARPER, ANNE
ADDRESS    : 12 WASHINGTON ST
             NEWTON          MA 02159

Enter Command (A-add P-print Q-quit): Q
OK,
```

DIRECT ACCESS FILES IN COBOL

COBOL treats direct access MIDASPLUS files as RELATIVE files. It uses the standard RELATIVE file I/O statements, with a few minor differences, as its interface to these type of files. The RELATIVE KEY in a RELATIVE file is the primary key that you defined for your MIDASPLUS file during template creation. Secondary keys have no counterpart in a RELATIVE file; they are not supported.

Note

In a direct access MIDASPLUS file, the records must be fixed-length; variable-length records are not supported.

COBOL requires that the RELATIVE KEY contain the relative record number. As a result, a special method exists for defining the primary key for direct access MIDASPLUS files to be accessed as COBOL RELATIVE files. The following sections explains this method.

Declaring the RELATIVE KEY in the Program

Use a corresponding PICTURE clause 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).

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. Declare the key in the WORKING STORAGE SECTION.

Relative files do not support secondary keys since COBOL does not provide a means for adding entries to secondary index subfiles.

Defining the File in a Program

The SELECT statement defines the file's logical name and organization. The ORGANIZATION clause specification and the terms used to describe the primary key are different for an INDEXED file than for a RELATIVE file. The format for these differences is

```
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. When accessing the file through a program, treat the RELATIVE KEY as a character string with a minimum size of one digit and a maximum size of eighteen digits. If the file has an access mode of SEQUENTIAL, it is not necessary to specify the key.

The following rules apply to RELATIVE KEY definition for RELATIVE files:

- The RELATIVE KEY must be an unsigned numeric integer.
- Do not specify a RELATIVE KEY with an OCCURS clause.
- The RELATIVE KEY must be the same length and type defined during template creation.
- The RELATIVE KEY cannot have a P character in its PICTURE clause.
- The RELATIVE KEY cannot be defined as a numeric with a separate sign.
- The RELATIVE KEY is numeric data defined as an ASCII or a bit string during template definition. It must represent the record number of each file record.
- The RELATIVE KEY cannot have a PICTURE clause larger than 9(18).
- The RELATIVE KEY that key-name-1 defines must be defined in the WORKING-STORAGE SECTION.

If the access mode is SEQUENTIAL, the RELATIVE KEY does not have to be defined within the program. If you do not define a key in the SELECT statement, the COBOL runtime library uses the size specified during template creation; the largest record number allowed is 999,999. The access modes are the same as those for INDEXED files.

See Appendix B, ERROR MESSAGES, for a list of the status codes returned in status-code.

ACCESSING RELATIVE FILES

COBOL uses the standard COBOL RELATIVE file interface to access MIDASPLUS direct access files. This method 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.

Opening and Closing the File

Use the OPEN statement to open a direct access file.

```
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, you cannot open a RELATIVE file for I-O if a WRITE statement is included for this file. It must be opened for OUTPUT only. Only empty files can be opened for SEQUENTIAL OUTPUT. (A WRITE is the only legal operation in OUTPUT mode.) COBOL provides its own record numbers in SEQUENTIAL WRITEs and ignores any values that you may have supplied for the relative record number field. See Table 6-1 for a list of statements that can be used in each access mode.

Use the CLOSE statement to close the file. For example,

```
CLOSE filename-1 [,filename-2...]
```

Put the CLOSE statement at the logical end of the program or in an error-handling routine. You can close more than one file with a single CLOSE statement.

Adding Records to a Relative File

COBOL's RELATIVE files do not support the use of secondary keys. To add a record to a direct access file, supply the record number and the data to be added to the data subfile.

Populating RELATIVE files is handled differently than populating INDEXED files. If a file contains no record or index subfile entries, you can add records to that file in any of the three access modes. Once a file contains entries, you can no longer add records to it in SEQUENTIAL access mode. The SEQUENTIAL access mode is intended for initial loading of records (that is, for populating empty files). Another restriction with SEQUENTIAL access mode is that the file must be opened for OUTPUT, not I/O. RANDOM and DYNAMIC access modes allow you to supply record numbers for each record to be added to the file.

The WRITE Statement: The WRITE statement adds records to a RELATIVE file in all three access modes. In SEQUENTIAL mode, COBOL supplies the record numbers for each record. You supply the data record information to an empty file. This method is called loading or initially loading a file. The file must be empty.

In SEQUENTIAL mode, the RELATIVE KEY starts at 1 and is incremented by 1 each time a new record is added. If you define the RELATIVE KEY field in the program, COBOL returns the record number of each record that you add after each WRITE operation is complete. This number is returned in the RELATIVE KEY field.

The random method is used with the RANDOM and DYNAMIC access modes. The random method requires you to supply a record number for each record added (slots are pre-allocated for the record numbers in the data subfile). Each record is placed in the proper slot according to its assigned record number.

Adding Records Randomly

In random WRITEs, you supply a record number for each record added. The format is the same as that shown for sequential writes. The file can be opened for OUTPUT or I-O. You can add entries in any order. Do not try to write a record that already exists and do not try to write beyond the preallocated file boundaries. If you only allocated 15 records, then do not try to add a record with number 000016 or above. CREATK asks for a number of records to allocate space for in an index subfile.

The format of the WRITE statement is:

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

record-name is the name of a record description associated with the file in the program's DATA DIVISION. The FROM identifier clause is optional. Without this clause, you must MOVE the new record information to record-name so that it can be written to the file. When the FROM identifier clause is used, data is moved from identifier to record name.

Note

START operations are not legal in RELATIVE files opened for SEQUENTIAL access in OUTPUT mode. The beginning of the file is the only place to start adding records.

Reading a Relative File

There are two types of READ formats for RELATIVE files: the sequential read format and the random read format. These formats are similar to those used in reading INDEXED files.

Sequential Reads: In a direct access file opened for SEQUENTIAL access, the READ operation retrieves records in order by record number. A MOVE and START command, or setting an initial value in WORKING-STORAGE, or a default, establishes the initial record number value. If the default is used, the initial position is set to the first record in the file. The first file record 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 clause moves the record read from the record buffer associated with the file into the read-var variable. The NEXT clause is used only in DYNAMIC access mode. This clause is not necessary if the file is opened for SEQUENTIAL access mode. A READ operation automatically causes the file pointer to move to the next record in the file. Always use the AT END

clause unless the program has a USE AFTER procedure for handling errors that occur while processing the file.

Reading the Current Record: In DYNAMIC access mode, READ without the KEY IS clause or the NEXT RECORD returns the current record. In RANDOM access mode, READ without the KEY IS clause also returns the current record. (Sequential reads are not possible in RANDOM mode.)

Notes

1. The current record is the record just read or positioned to by a START operation. (START is not legal in RANDOM access).

2. In SEQUENTIAL access mode, the file pointer is advanced automatically to the next record in the file each time that a read statement is encountered.

Keyed Reads: Keyed reads (random reads) are permitted in DYNAMIC and RANDOM access modes, if you use the RELATIVE KEY. To do a keyed read, 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].
```

Use the INTO clause to move the data record from the buffer into which it is read to a program-specified temporary storage area specified by read-var. Unless a USE AFTER procedure under the DECLARATIVES specifies what to do when errors occur during processing of the file, the INVALID KEY clause is required.

Move a new value to the RELATIVE KEY field before each READ or the same record is returned repeatedly. The only way to do a keyed READ on a RELATIVE file is to supply a record number value for the RELATIVE KEY field.

Updating Records

The REWRITE statement replaces the current record with a new text string and destroys the original. REWRITE does not establish or change file position. Do a READ before a REWRITE in all access modes in order to tell MIDASPLUS which record is to be updated and to lock the record. In order to update the file, open it for I-O.

The REWRITE format is:

```
REWRITE record-name [FROM identifier]
[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 clause is optional. You can move the new record value to record-name before you do the REWRITE. Unless the DECLARATIVES include a USE AFTER procedure for dealing with errors that occur while processing this file, the INVALID KEY clause is required in RANDOM and DYNAMIC modes. When the file is opened for SEQUENTIAL access, do not include the INVALID KEY clause in REWRITE statements.

Deleting Records

The DELETE statement removes an indicated record from a direct access file. You can do deletes in RANDOM, DYNAMIC, or SEQUENTIAL access modes as long as the file is opened for I-O. The DELETE format is:

```
DELETE filename RECORD [INVALID KEY imperative-statement].
```

Unless a USE AFTER procedure for trapping errors is included under the DECLARATIVES, use the INVALID KEY clause in RANDOM and DYNAMIC modes. The INVALID KEY clause is not legal in SEQUENTIAL access. A READ statement, and not the DELETE, establishes the record to be deleted.

Deletes in SEQUENTIAL Mode: In SEQUENTIAL access mode, the record to be deleted must be READ before a DELETE can be executed. The DELETE statement in SEQUENTIAL access mode cannot establish a current record position and depends on a READ statement to do so. Do not include the INVALID KEY clause 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 required to READ prior to a DELETE since a DELETE establishes the current record position on its own. Supply a value for the RELATIVE KEY before a DELETE, which is then used to position to that record before deleting it.

RELATIVE PROGRAMMING EXAMPLE

The following RELATIVE program adds names to the direct access file (DACUST) that was created in Chapter 2.

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID.       CUST.
       INSTALLATION.     PRIME COMPUTER, INC.
       DATE-WRITTEN.     05/13/85.
       DATE-COMPILED.
       REMARKS.          ADDS RECORD TO A DIRECT ACCESS FILE.

       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       SOURCE-COMPUTER.  PRIME-750.
       OBJECT-COMPUTER.  PRIME-750.
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT CUST-FILE ASSIGN TO PFMS
                  ORGANIZATION IS RELATIVE
                  ACCESS MODE IS DYNAMIC
                  FILE STATUS IS STATUS-CODE
                  RELATIVE KEY IS ACCT-NO.
      *
       DATA DIVISION.
       FILE SECTION.
       FD  CUST-FILE
           VALUE OF FILE-ID IS 'DACUST'.
       01  CUST-RECORD.
           05  CUST-ID          PIC X(5).
           05  CUST-NAME        PIC X(25).
           05  LOCATION-CODE    PIC X(4).
       WORKING-STORAGE SECTION.

       01  ACCT-NO              PIC 9(6).
       01  X-ACCT-NO            PIC X(6).
       01  J-ACCT-NO            PIC X(6) JUSTIFIED RIGHT.
       01  ANS                  PIC X.
       01  EOF                  PIC X.
       01  STATUS-CODE          PIC XX.
       PROCEDURE DIVISION.
       MAINLINE-CONTROL.
           OPEN I-O CUST-FILE.
           PERFORM GET-RECORDS THRU GR-EXIT
                   UNTIL ANS = 'N' OR ANS = 'n'.
           CLOSE CUST-FILE.
           DISPLAY 'Print file? ' WITH NO ADVANCING.
           ACCEPT ANS.
           IF ANS = 'Y' or 'y'
              PERFORM PRINT-FILE.
           STOP RUN.

       GET-RECORDS.
           MOVE SPACES TO CUST-RECORD, X-ACCT-NO, ACCT-NO.
```

```
           DISPLAY 'Enter account number: '
                   WITH NO ADVANCING.
           ACCEPT X-ACCT-NO.
           UNSTRING X-ACCT-NO DELIMITED BY SPACE INTO J-ACCT-NO.
           INSPECT J-ACCT-NO REPLACING LEADING SPACES BY ZEROES.
           IF J-ACCT-NO IS NUMERIC
              MOVE J-ACCT-NO TO ACCT-NO
           ELSE
              DISPLAY 'Error: invalid account number ,' X-ACCT-NO
              GO TO GR-EXIT.

           DISPLAY 'Enter customer id (5 chars): '
                   WITH NO ADVANCING.
           ACCEPT CUST-ID.
           DISPLAY 'Enter Customer Name (25 chars): '
                   WITH NO ADVANCING.
           ACCEPT CUST-NAME.
           DISPLAY 'Enter Location Code (4 chars): '
                   WITH NO ADVANCING.
           ACCEPT LOCATION-CODE.
           WRITE CUST-RECORD
                 INVALID KEY
                      PERFORM WRITE-ERR
                      GO TO GR-EXIT.

           DISPLAY 'Account number ' ACCT-NO ' added'.
           MOVE SPACE TO ANS.
           DISPLAY 'More? (Y/N) ' WITH NO ADVANCING.
           ACCEPT ANS.

       GR-EXIT.
           EXIT.

       WRITE-ERR.
           IF STATUS-CODE = 22
              DISPLAY 'Error: customer already exists ', ACCT-NO
           ELSE
              DISPLAY 'Error writing to customer file: ', STATUS-CODE.

       PRINT-FILE.
            OPEN INPUT CUST-FILE.
            READ CUST-FILE NEXT RECORD
                 AT END MOVE 'Y' TO EOF.
            PERFORM READ-PRINT UNTIL EOF = 'Y'.
            CLOSE CUST-FILE.

       READ-PRINT.
            DISPLAY ACCT-NO, ' ' CUST-ID ' ' CUST-NAME ' '
                 LOCATION-CODE.
            READ CUST-FILE NEXT
                 AT END MOVE 'Y' TO EOF.
OK, CBL CUST

[CBL rev 19.4]
```

```
OK, BIND
[BIND rev 19.4.1]
: LOAD CUST
: LI CBLLIB
: LI
BIND COMPLETE
: FILE
OK, RESUME CUST
Enter account number: 1
Enter customer id (5 chars): 1234A
Enter Customer Name (25 chars): HARPER, ANNE
Enter Location Code (4 chars): NEMA
Account number         1 added
More? (Y/N) Y
Enter account number: 5
Enter customer id (5 chars): 5675B
Enter Customer Name (25 chars): CORRADO, THOMAS
Enter Location Code (4 chars): SWMA
Account number         5 added
More? (Y/N) N
Print file? Y
        1 1234A HARPER, ANNE                NEMA
        5 5675B CORRADO, THOMAS             SWMA
OK,
```

7
The BASIC/VM Interface

The BASIC/VM interface to MIDASPLUS consists of a special set of file handling statements that read, write, delete, and update entries in a MIDASPLUS file. The MIDASPLUS access statements are similar in format to the standard file handling statements in BASIC/VM. Extensions to their syntax permit the more complex operations associated with MIDASPLUS files. The interface acts as a go-between and translates your demands into the MIDASPLUS subroutine calls that do all of the work. As a result, MIDASPLUS files can be processed as easily as any other file type that BASIC/VM supports.

This chapter discusses the BASIC/VM language dependencies, access statements, and error handling, and provides programming examples.

LANGUAGE DEPENDENCIES

As all of the other language interfaces (except PL/I), MIDASPLUS requires you to use CREATK to create a template before BASIC/VM can access a MIDASPLUS file. In addition, the following considerations apply to the BASIC/VM MIDASPLUS interface:

- Only keyed-index access MIDASPLUS files are supported. (BASIC/VM cannot process direct access MIDASPLUS files.)
- Up to 17 secondary indexes are allowed per file (duplicate index entries are allowed).

- Although key fields are not required to be part of the data record, it is strongly recommended that you include them.
- The secondary data feature is not supported.

BASIC/VM refers to keys by number. The primary key is KEY0, the first secondary key is KEY01, the second is KEY02. Key numbers and index subfile numbers are synonymous. A reference to a particular key is also a reference to the index subfile in which the key values are stored.

SUMMARY OF ACCESS STATEMENTS

Table 7-1 summarizes the BASIC/VM statements needed to process MIDASPLUS files. These statements are similar to those used to handle other BASIC/VM file types.

Table 7-1
Summary of Access Statements

Statement	Function
DEFINE FILE	Opens the designated MIDASPLUS file on an available PRIMOS file unit. Assigns a user-specified 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 that record.
REWIND	Positions the file pointer to the first entry in the specified index. Defaults to primary index.
ADD	Adds a record to the MIDSAPLUS file in primary key sequence. You may not include secondary keys with the ADD command.
READ	Finds, locks, and returns a MIDASPLUS file record by primary or secondary key. Other READ options allow duplicate retrieval as well as sequential record retrieval.
UPDATE	Rewrites the current record.
REMOVE	Deletes a record by primary key. Can also delete any secondary index entry.

LOCKING AND UNLOCKING RECORDS

BASIC/VM has no specific lock or unlock statements. In order to keep the integrity of any record, the READ, UPDATE, POSITION, and DELETE statements all lock a record before they perform their operations. BASIC/VM locks the record to protect it from accidental harm by another user or process. Include error traps in your programs for records that other users may already have locked.

OPENING/CLOSING A MIDASPLUS FILE

BASIC/VM opens and closes a MIDASPLUS file the same way it does any other file, by means of the DEFINE FILE and the CLOSE statements respectively.

The DEFINE FILE Statement

The DEFINE FILE statement opens a MIDASPLUS file and assigns it a file unit number. This number is used as an alias for the file during the remainder of the program. You can open up to 12 files at a time from a single BASIC program. Be sure to use the unit numbers 1 to 12 when opening a MIDASPLUS file. The keyword MIDAS is required. The format of this statement is:

```
DEFINE [READ] FILE #unit = filename, MIDAS [,record-size]
```

#unit is the user-assigned unit number (either a literal or numeric expression). The # sign is required, as in: #2.

filename is the name of the MIDASPLUS file. The name can either be a BASIC string variable that contains the filename or the filename can be a quoted constant string.

record-size is the length of the MIDASPLUS data subfile record in 16-bit units. If the MIDASPLUS file has fixed-length records, this number, if specified, must match the data size indicated in the MIDASPLUS template. (Use the PRINT option of CREATK to determine this.) If the file has variable-length records, no record-size is necessary. The default record-size is 60 and the parameter is optional.

The READ option opens the file for reading only and does not allow you to add records to the file. The default access mode (when you omit the READ option) allows the full range of file operations to be performed on the MIDASPLUS file without restrictions.

For example, the first of the following two statements opens a file with fixed-length records. The second opens a file with variable-length records.

```
DEFINE FILE #1 = 'BANK', MIDAS, 35
DEFINE FILE #1 = 'VARBANK', MIDAS
```

CLOSE Statement

MIDASPLUS files are closed just like any other BASIC/VM files:

```
CLOSE #unit
```

#unit is the user-assigned BASIC/VM file unit on which the MIDASPLUS file is opened and specified in the DEFINE FILE statement. The # sign is required.

ERROR HANDLING

The BASIC/VM ON ERROR statement directs program control to a statement that will be executed if an error occurs. This action traps MIDASPLUS errors. ON ERROR traps I/O errors on the particular unit on which the file was opened or on all file units with open files (a general error trap).

Following an error, use MIDASERR, which prints the value of the MIDASPLUS error code. This works much like ERR, the special error-code variable that returns BASIC/VM error codes. Look up the MIDASPLUS error code in Appendix B, ERROR MESSAGES, 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 MIDASPLUS file was opened. If #unit is not specified, all I/O errors occurring on every opened file unit are trapped.

Line-number is the line number of the first statement of the error handler.

Use the following PRINT statement to print out the MIDASPLUS error code.

```
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 the record in the file to which the file pointer is pointing and is established relative to an index subfile. The file pointer points to a specific entry in an index subfile. This index subfile entry in turn points to a record in the data subfile. This record is known as the current record.

You can establish the file position in the primary index subfile or in one of the secondary index subfiles with the REWIND, POSITION, or READ statements. If you do not supply a key value or an index number, the file position uses the primary index by default. The current record is then set to the record referenced by the first entry in the primary index subfile. The index subfile that points to the current record is the current index.

The POSITION Statement

The POSITION statement moves the file pointer to an index subfile entry for any specified record in the MIDASPLUS file, making that record the current record. POSITION locks the record and leaves it locked until the file pointer is positioned to another record. The format is:

```
POSITION #unit { SEQ                                  }
               { ,KEY [key-number] = key-value        }
               { SAME KEY                             }
```

key-number is the key number (index subfile number) that may be a literal or numeric expression. If it is unspecified or zero, it is taken as the primary key.

key-value is the key value enclosed in quotes or a legal string expression. Use this option to make a specific record current.

SEQ positions the pointer to the next sequential record in the file according to the order of entries in the current index.

SAME KEY positions the pointer to the next record with the same key value as the current record. Use this option when a secondary key allowing duplicates establishes the most recent file position.

If there is no record at the specified file position, an error is 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 according to primary or secondary key as they are specified on the POSITION statement line. This action establishes that key as the current key of reference. It also establishes the index subfile in which the 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 specifically 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.

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. REWIND sets the file pointer relative to the first entry in the indicated index subfile. (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. Use it with the KEY option.

Examples

The first example sets the file position to the record referenced by the initial entry in the primary index. The second example sets the position to the record referenced by the first entry in secondary index subfile 03.

```
REWIND #3
REWIND #2, KEY 3
```

ADDING RECORDS

The ADD statement adds a record to a MIDASPLUS file without changing 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. This practice is recommended, because it avoids the possibility of having index entries that do not match the key values in the data record. Whenever keys are stored in the data record, make sure that the entries in the primary and secondary index subfiles are the same as the key entries stored in the data subfile record. This action minimizes confusion. The format of the ADD statement is:

ADD #unit, new-record, { PRIMKEY | KEY[0-expr] = key0-val [,keylist] }

keylist has the form:

KEY key-number = key-val ...

You can repeat the above expresson for each secondary key field in the record. If you are storing keys in the data record, the key values assigned here should match the key values in the data record, specified by the new-record. The parameters are described below:

new-record is the data record to be added. If the file has fixed-length records, new-record should be equal in length to the record size declared for the file. Pad the record to the correct length with blanks. If you want keys stored in the record, make sure new-record includes all of the key values.

PRIMKEY represents the primary key.

KEY[0-expr] represents the primary key. 0-expr is a literal or numeric expression that evaluates to zero.

key0-val represents the primary key value. It may be a string expression or a literal.

keylist is an optional list of secondary key numbers and values.

key-number is a numeric expression indicating a secondary key number (index subfile number).

key-val is a string expression or literal containing a secondary key value.

Only the keys that 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 confusion. 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

The following example shows a BASIC/VM program that adds records to the BANK file that was created in Chapter 2. The program reads data from a sequential disk file, called NAMES, pads each record with blanks until the record is the correct length, and then adds each record to the data subfile. (Since BASIC treats commas as delimiters, you cannot include commas in your input file.) Since all of the necessary key values were included in the keylist, the primary and secondary key entries are placed in the proper index subfiles at the same time. This example also shows the output from the program when it is executed. The program is run from the PRIMOS command level, using the BASICV command.

```
10 DEFINE FILE #1 = 'BANK', MIDAS, 43
20 DEFINE FILE #2 = 'NAMES', 22
30    DEF FNS$(X$,N)  ! PADS STRING X$ WITH SPACES
40      ! ADD SPACES UNTIL THE LENGTH EQUALS 86 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 = 4
90 N = 86 ! 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$,1,9)
140    PRINT 'PRIMARY KEY IS:': B$
160    C$ = SUB(A$,10,34)
170    D$ =SUB(A$,35,44)
180    ! PAD STUFF HERE
190    A$ = 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
RECORD VALUE IS: 276503889harper  anne              chk4123891
PRIMARY KEY IS: 276503889
RECORD VALUE IS: 036792406harper  anne              ln72537465
PRIMARY KEY IS: 036792406
RECORD VALUE IS: 189264289murray  paul              mc28374646
PRIMARY KEY IS: 189264289
RECORD VALUE IS: 023677386corrado  thomas           sav1273565
PRIMARY KEY IS: 023677386
DONE
OK,
```

Note the method used in the above example to pad the data record to the specified record length. This method is mandatory in fixed-length record files or you will get an error message when you attempt to add the record.

READING RECORDS

The BASIC/VM READ statement provides you with a complete range of options for getting information from a MIDASPLUS file. You can read a MIDASPLUS 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, which guards against concurrency errors. The READ statement format allows you to read records from a MIDASPLUS file sequentially, by duplicate keys, and by primary or secondary key values. When the READ is complete, the record remains locked until another operation is performed to change the file pointer location.

READ Options

The READ format is:

$$\text{READ [KEY] \#unit} \left\{ \begin{array}{l} \text{SEQ} \\ \text{,[KEY [key-num] = key-val]} \\ \text{SAMEKEY} \end{array} \right. \text{,read-var} \left. \right\}$$

The keywords and parameters used in this format are described below:

[KEY] is used for reading the full value of any key contained in an index subfile (optional).

#unit is the user-assigned unit number on which the file is opened. The # sign is required.

KEY is used with key-num and key-val to indicate which record should be read. If you omit the KEY [key-num] = key-val clause, the current record is read. In this case, a POSITION or REWIND must establish the current record before you specify the READ statement.

key-num is a literal or numeric expression indicating which key (index subfile) should be used in this retrieval. If omitted, the default is 0 (the primary key).

key-val is the full or partial value of the key on which to conduct the search. It is used in conjunction with KEY [key-num].

SEQ indicates that the next sequential record, as determined by the current index of reference, should be read.

SAMEKEY reads the next record with the same key value as the current record.

read-var is a string variable into which the retrieved record value is read.

READ Examples

The following program uses the SAMEKEY option to find all of the records with the same secondary key value.

```
10 DEFINE FILE #1 = 'BANK',MIDAS,43
20 ! THIS PROGRAM FINDS FIRST OCCURRENCE OF A CERTAIN
30 ! SECONDARY KEY VALUE AND THEN FINDS ALL THE DUPLICATES
40 READ #1,KEY01='harper  anne', 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 VALUE
100 PRINT 'RECORDS WITH DUPLICATE VALUES ARE:'
110 PRINT
120   FOR I = 1 TO 4
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
FIRST RECORD WITH THIS VALUE:
276503889harper  anne              chk4123891


RECORDS WITH DUPLICATE VALUES ARE:

036792406harper  anne              ln72537465

BASIC ERROR IS:       RECORD NOT FOUND
ER!
```

This program only works when secondary keys allow duplicates. The duplicate feature is turned on or off during template creation. The error handler is used to trap the error that will occur when you do not find any more duplicates for this key value.

Partial key values can be used in a READ statement, as in:

```
    READ #1, KEY 1 = 'Har', K$
```

The full value of this key is actually "Harper Anne". Only the left most positions of a key value are allowed as partial key values. It would be illegal to do a search on "Anne" in this case.

As stated above, the READ KEY option returns the full value of a key. With READ KEY, you can find out what key you are currently positioned on (that is, the index you are using as the index of reference) or you can get the full key value of any key by specifying a partial key value. For example:

```
10 DEFINE FILE #1 = 'BANK', MIDAS,43
20 ! READ RECORD WITH PARTIAL KEY
30 READ #1, KEY 1 = 'corra', 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 = 'sav', 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
RECORD READ WITH PARTIAL KEY:
023677386corrado  thomas          savl273565

CURRENT KEY VALUE IS:
corrado  thomas

FULL KEY :
savl273565

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. This means that you should not attempt to change key values with UPDATE. To change key values, 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. This 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 the record to be modified. However, you can use a REWIND or POSITION statement to establish the current record position instead of READ. Because UPDATE overwrites the existing record rather than deleting and replacing it, make the new record equal in length to the old one to ensure that all of the old data is completely overwritten. For example:

```
10 DEFINE FILE #1 = 'BANK', MIDAS,43
20 ! FIND RECORD TO BE UPDATED
30 READ #1, KEY1='corr', X$
40 PRINT 'ORIGINAL RECORD IS:', X$
50 PRINT
60 ! UPDATE THIS RECORD BY ADDING SOMETHING TO THE END
70 A$= ' Call before 11:30 A.M.'
80 ! WRITE THE ORIGINAL RECORD BACK WITH THIS ADDITION
90 X$ = SUB(X$,1,43) ! TAKE JUST THE NON-BLANK PART
100 X$ = X$+A$ ! COMBINE THE TWO
110 ! NOW PAD TO CORRECT LENGTH
120 X$ = X$ +' ' UNTIL LEN(X$) = 86
```

```
130 UPDATE #1,X$
140 REWIND #1
150 READ #1, KEY='023677386', X$
160 PRINT 'UPDATED RECORD:', X$
170 CLOSE #1
180 END
OK, basicv update
ORIGINAL RECORD IS:
023677386corrado  thomas          sav1273565


UPDATED RECORD:
023677386corrado  thomas          sav127356 Call before 11:30 A.M.

OK,
```

In files with fixed-length records, pad the record to the correct length; otherwise you will get a record-size error and the update will fail.

DELETING RECORDS

The REMOVE statement selectively deletes index entries for a particular data record. If you specify only the primary key, the associated data record and 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. The REMOVE format is:

```
REMOVE #unit [,KEY [key-num] = key-val]
     [,KEY [key-num]=key-val]...
```

key-num is the 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 the 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 REMOVE without the options.

DELETE Example

The following example shows how to remove specific secondary key values from an index and how to delete an entire record and its primary index entry. Note that removing values from a secondary index does not change the data record. The program uses the MIDASERR feature to print out the MIDASPLUS error code associated with the read error that occurs on an attempt to read a deleted record.

```
10 DEFINE FILE #1 = 'BANK', MIDAS,43
20 ! REMOVE A SECONDARY INDEX ENTRY FROM THIS FILE
30 PRINT 'REMOVE THE SECONDARY KEY VALUE: murray  paul'
40 REMOVE #1, KEY01='murray  paul'
50 ! BUT THE RECORD VALUE REMAINS UNCHANGED
60 PRINT
70 READ #1, KEY = '189264289', 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 4
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='189264289'
270 REWIND #1
280 !
290   ON ERROR GOTO 340
300 PRINT
310 READ #1, KEY='189264289', X$
320 PRINT X$
330 GOTO 360 ! IF NO ERROR
340 PRINT 'MIDASPLUS ERROR CODE:': MIDASERR
350 PRINT 'BASIC ERROR IS:': ERR$(ERR)
360 CLOSE #1
370 END
OK, basicv delete
REMOVE THE SECONDARY KEY VALUE: murray  paul

RECORD VALUE IS:
189264289murray  paul              mc28374646
```

```
RECORDS READ BY SECONDARY KEY:

276503889harper  anne              chk4123891

036792406harper  anne              ln72537465


NOTE: RECORD REFERENCED BY DELETED
INDEX ENTRY IS NOT PRINTED

DELETE RECORD BY PRIMARY KEY

MIDASPLUS ERROR CODE: 7
BASIC ERROR IS: RECORD NOT FOUND
ER!
```

The MIDASPLUS error code of 7 indicates an unsuccessful read resulting from a failure to find a record with the indicated key value.

8

The PLI Interface

This chapter documents the PL/I interface to MIDASPLUS files. PL/I views a MIDASPLUS file as a RECORD KEYED SEQUENTIAL file that the standard PL/I READ, WRITE, REWRITE, and DELETE statements can access. PL/I requires that the MIDASPLUS file have an ASCII primary key. Besides supporting CREATK-defined files, PL/I can create its own MIDASPLUS 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 MIDASPLUS only. See the PL/I Reference Guide for complete syntax information on these and other PL/I statements referenced here.

Language Limitations

The PL/I interface to MIDASPLUS does not support the following MIDASPLUS features:

- Non-ASCII primary keys
- Secondary keys
- Direct access MIDASPLUS files
- Secondary data

Since PL/I does not support secondary keys or non-ASCII primary keys, you cannot use PL/I to set up a MIDASPLUS file template with these features. To create a MIDASPLUS file with secondary keys, fixed-length records or a primary key of less than 32 characters, use CREATK. You can access this type of file from a PL/I program as long as its primary key is a character string of 32 characters or less. You can still access existing MIDASPLUS files with secondary keys from a PL/I program, but you will not be able to access any secondary index subfiles from PL/I.

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, you can access the file from PL/I using character or numeric key values. Numeric values are converted to character strings according to the standard PL/I conversion rules.

RUNNING A PL/I PROGRAM

The following is a sample compile and load sequence using BIND. User input is underlined to distinguish it from system output. Substitute the name of your program for the program argument in the following example. This example appears in uppercase to help you distinguish between the letter l and the number 1. You may use either uppercase or lowercase letters.

```
OK, PL1 program-name.PL1
[PL1 Rev. T1.0-21.0 Copyright (c) 1986, Prime Computer, Inc.]
0000 ERRORS [PL1 Rev. 22.0]

OK, BIND
[BIND Rev. 21.0.1 Copyright (c) 1985, Prime Computer, Inc.]
: LOAD program-name
: LI PL1LIB
: LI
BIND COMPLETE
: FILE

OK, RESUME program-name
```

OPENING/CREATING A MIDASPLUS FILE

There are two ways to create a MIDASPLUS file for use with PL/I. You can create the file with CREATK and then open it for reading and/or writing from a PL/I program. You could also create a MIDASPLUS file from PL/I using the standard file I/O statements. The next part of this chapter tells how to create a new file with PL/I and how to open an existing MIDASPLUS file.

For information on opening non-PL/I created files (MIDASPLUS files created directly with CREATK or KX$CRE), see ACCESSING CREATK-DEFINED FILES below.

Creating A MIDASPLUS File From PL/I

When creating a MIDASPLUS file from a PL/I program, declare the file explicitly with the KEYED SEQUENTIAL [RECORD] attributes. It is not necessary to specify the RECORD attribute since the KEYED attribute implies it. Follow this sequence when creating a MIDASPLUS file from PL/I:

```
DECLARE filename FILE KEYED SEQUENTIAL;
OPEN FILE(filename) OUTPUT;
```

The FILE keyword in the DECLARE statement must appear before or after the actual filename. You can abbreviate the DECLARE statement to DCL. These statements tell MIDASPLUS to open a MIDASPLUS file with the default attributes. (See The PL/I Reference Guide for details on OPEN and DECLARE syntax and attributes.) The MIDASPLUS file will have variable-length records with a maximum length of 4096 bytes and a maximum primary key of 32 characters.

Although the default is defined as an ASCII character string, PL/I conversion rules allow you to define the primary key from program level

as a character or numeric item. You can write file entries with character or numeric key values. Remember that the data must be read in the same format in which it was written. You cannot mix and match data types in WRITE and READ statements.

Opening an Existing MIDASPLUS File

To open an existing MIDASPLUS file that was previously created with PL/I, use the open statement with this format:

```
OPEN FILE(filename) { INPUT  }
                    { OUTPUT }
                    { UPDATE }
```

Filename must not be longer than 8 characters. You may specify it before or after the FILE keyword. Declare the file as KEYED SEQUENTIAL. The INPUT, OUTPUT, and UPDATE options indicate the access mode in which to open the file. The access mode controls the types of operations that can be performed on the file.

The INPUT option allows READ operations only and 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. If such a file does not exist, PL/I creates a new one. Read and 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

You can include the INPUT, OUTPUT, or UPDATE options in the DECLARE statement where the KEYED SEQUENTIAL file is first declared. This eliminates the need for an explicit OPEN statement, for example:

```
DECLARE SAMPLE FILE KEYED SEQUENTIAL OUTPUT;
                    or
DCL OTHER FILE KEYED SEQUENTIAL INPUT;
```

A file opened or declared as INPUT or UPDATE must already exist. If it does not exist, an error is returned.

FILE I/O CONCEPTS IN PL/I

The following PL/I statements are all used to access MIDASPLUS files from PL/I programs:

Statement	Function
READ [KEY]	Reads next sequential record in a file (if KEY is 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).

Table 8-1 indicates which statements can be used in each access mode.

Table 8-1
Access Mode Statements

INPUT	OUTPUT	UPDATE
READ	WRITE	READ WRITE REWRITE DELETE

The Current Record in PL/I

PL/I automatically handles the current file position. As a result, you do not have to be concerned about a communications array to keep track of the current interface. Before performing the read operation, the PL/I READ statement advances the current file position pointer to the next sequential record in the file. 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 that it can insert the primary key value of the record being added in its proper place. The current record after a WRITE is the record just

written. No record is current after a DELETE because DELETE always removes the current record. REWRITE does not update the current record position.

Initial Current Record: When a MIDASPLUS file is opened for INPUT or UPDATE in PL/I, the current file pointer is set just before the first record in the file. A READ (with or without KEY) establishes a current record position. This establishes the current file position and initializes the MIDASPLUS communications array that PL/I transparently handles for you.

DELETE and the Current Record: After a DELETE operation, the current record is left undefined until the next READ or WRITE operation. Unless you just deleted the last record in the file, a sequential READ (without the KEY option) will work. 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.

Locked Records

Since the PL/I WRITE 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 problems if you hit CTRL-P or BREAK immediately after a WRITE or REWRITE operation. Records are locked only for WRITE or REWRITE statements.

ADDING RECORDS

To add records to a new or existing file, open the file for OUTPUT or UPDATE. Use the following PL/I WRITE statement to write records to the MIDASPLUS file.

```
WRITE FILE(filename) FROM(var) KEYFROM(keyvar);
```

The var argument contains the new record information. Declare its data type as CHARACTER and its size (which PL/I views as the record size) as varying (VAR) if desired. (See Declaring Data Size Section below.)

The KEYFROM Option

The KEYFROM option specifies the primary key value for this new record. Make sure that KEYFROM and a unique key value are present in every WRITE statement performed on a MIDASPLUS file. You can declare keyvar as a character string of 32 characters or less, or as a numeric field (for example, fixed binary or fixed decimal). If declared as a CHARACTER, it cannot be declared as VARYING. If the template was created with CREATK, make sure that the variable matches the size and type of the primary key as defined during template creation.

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 is recommended that you declare the key variable as 32 characters even if you have fewer characters. For example, if you have a 10 character key, declare the key variable as 32 characters. It is better to have PL/I add the remaining 22 characters as blanks than to have unpredictable values stored in the index entry slot.

Since PL/I uses only primary keys, no duplicates are allowed. If the key value specified for keyvar already exists in the file, a KEY error will be reported and the program will halt. Make sure the primary keys are unique. Supply a new value for the keyvar argument with every WRITE statement. (See the Error Handling section of this chapter for more information on KEY errors.)

The WRITE Operation

The file must be opened for I/O or OUTPUT in order to use the WRITE statement. WRITE updates the current record position in order to add a new record to the file. WRITE positions the file to the proper index location. After a WRITE, the current record is the one just written. A read next record operation after a WRITE returns the record immediately following the one just added.

Each WRITE operation places the 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. When reading sequentially through the file, you will get all of the records in the order that you expect them (based on primary index entry order) rather than in the order in which you added them.

Declaring Data Size

It is not possible to create a true fixed-length record MIDASPLUS file from a PL/I program. MIDASPLUS requires you to declare a maximum size for the data variable from which each MIDASPLUS file record will be written. This is the variable that your program puts data record

information into so it can be written to the MIDASPLUS file as a unit. By setting the size of this variable, you effectively limit the record size of the MIDASPLUS file. For example:

```
DECLARE BANK FILE KEYED SEQUENTIAL;
DECLARE PKEY CHAR(32);
DECLARE DATAVAR CHAR(30);
PKEY = 'aaaa';
WRITE FILE(BANK) FROM(DATAVAR) KEYFROM(PKEY);
```

In the above example, datavar is set to 30 characters, indicating that the records written to the MIDASPLUS file BANK 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 (the maximum key size that PL/I allows).

Example: The OPENIT program, listed below, opens a new MIDASPLUS file called Sample and adds records to it. Since pkey is declared as char(32), primary key values are supplied in ASCII form. The primary key can either be a character string or a fixed decimal.

```
openit:
        proc options(main);

/* This program creates a MIDASPLUS file called Sample */

dcl sample file keyed sequential;              /* MIDASPLUS */
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 keep data integrity and to allow for future file requirements, include the primary key in the data record as shown below:

```
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, use PUT EDIT to print out just the part of the record that you want. The key can also be stored at the end of the data record. Since the primary key entry in the index subfile cannot be changed, avoid making changes to the key field of the record during an update.

READING A MIDASPLUS FILE

There are three types of file reads that PL/I can perform on a MIDASPLUS file:

- Keyed read - a record is found based on a user-supplied key value.
- Sequential read (also called non-keyed read) - records are read from the file in primary key order. Either a keyed read or the default establishes file position at some point in the primary index subfile. This action puts the file pointer at the beginning of the index subfile.
- Reading key values - the full key value of the primary key, as MIDASPLUS stores it in the primary index, is returned. (This 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: open the file for INPUT or UPDATE in order to read it. The READ statement format is:

```
READ FILE (filename) INTO(var) [KEY(keyvar)] [KEYTO (curkey)];
```

A record is read into the var variable. Declare the record according to the following rules.

For PL/I-created files, match var in size and type with the WRITE argument that was used in writing the 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 READ statement.

For non-PL/I created MIDASPLUS files with fixed-length records, use the fixed record size value in declaring the INTO argument. (See Reading Fixed-Length Records below.)

For variable-length records in a file that PL/I did not create, make sure that the record size is not larger than 256 bytes.

The KEY and KEYTO Options

The KEY option finds the record whose key matches the one specified in keyvar. KEY is used in keyed reads only. (See Keyed Reads below.) If KEY is omitted, the next sequential record is read. Make sure the keyvar argument matches 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 would 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.) Always declare the curkey argument as CHAR(32) VARYING, because of the way that PL/I handles the MIDASPLUS index subfile entries.

Note

The KEY and KEYTO options cannot appear together in the same READ statement. Use KEY for keyed reads when the key value of a record is known. Use KEYTO during sequential reads when you want to determine the primary key value of the current record. KEYTO is especially useful when keys are not stored in the actual data file record.

Keyed Reads

To perform keyed reads, specify a valid key value with the KEY option. Valid means that the value must occur in the primary index subfile of the MIDASPLUS file being read. Key clauses that do not appear in the MIDASPLUS 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

Use a READ without the KEY option to read a MIDASPLUS file sequentially, in primary key order. The current record pointer 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. The following is an example of a sequential file read:

```
ON ENDFILE(SAMPLE) GOTO CLOSE_FILES;
DO WHILE('1'B);  /* infinite loop */
READ FILE(SAMPLE) INTO(READVAR);
PUT SKIP LIST(READVAR);
END;
```

All of the records in the file from the current record are read as the program loops. Then, the ENDFILE condition is signaled and control is sent to the part of the program labeled "CLOSE_FILES" where the file is closed.

Reading Key Values

You can use the KEYTO option to obtain the value of the primary key of the record currently being read. For example:

```
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 PL/I PUT EDIT statement is useful when you want PL/I to print only the first few characters of the primary index subfile entry. Without this statement, PL/I prints a 32-character version of the primary key.

Below is a listing of a sample program that performs keyed and sequential reads on the file that OPENIT created.

```
OK, SLIST READ.PL1
read:
        proc options(main);
dcl sample file keyed sequential;
dcl pkey char(32);                              /* primary key */
dcl sysprint file;
dcl readvar char(30) 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;

OK, PL1 READ.PL1
[PL1 Rev. T1.0-21.0 Copyright (c) 1986, Prime Computer, Inc.]
0000 ERRORS [PL1 Rev. 22.0]

OK, BIND
[BIND Rev. 21.0.1 Copyright (c) 1985, Prime Computer, inc.]
: LOAD READ
: LI PL1LIB
: FILE
: LI
BIND COMPLETE
: FILE

OK, RESUME READ

first file record
second file record
third file record
Key value:  0003
```

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 a user-specified record with a new record value.

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. The current record position is not updated after the REWRITE operation is complete.

The REWRITE Statement

Records in an existing MIDASPLUS file can be updated with the REWRITE statement:

```
REWRITE FILE(filename) FROM(datavar) [KEY(keyvar)];
```

Datavar is the variable containing the data that will replace the record being updated. You cannot update just part of a record. You must rewrite the whole thing. If you make a mistake while adding the original key field, delete the record and then WRITE it over correctly.

The REWRITE KEY Option

Use the KEY option when you want to specify exactly which record will be updated, to avoid confusion about the record being updated. Without the KEY option, REWRITE updates the current record, which is the record that the READ statement just read or the record that the last WRITE statement wrote when no intermediate READ statement occurred. This means that a current record position would have already been established (by a READ, WRITE, or another REWRITE with the KEY option) before a REWRITE without the KEY option. In this case, PL/I uses the MIDASPLUS communication array to determine where the current record is and which record should be updated. If the KEY option is used with REWRITE, it is not necessary to perform a READ or WRITE before REWRITE. If keyvar indicates a key value that does not exist in the file, a key error is triggered and the program aborts.

The following program performs an update on the MIDASPLUS file SAMPLE.

```
OK, SLIST UPDATE.PL1
update:
        proc options(main);
dcl sample file keyed sequential;
dcl pkey char(32);                              /* primary key */
dcl sysprint file;
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, PL1 UPDATE.PL1
[PL1 Rev. T1.0-21.0 Copyright (c) 1986, Prime Computer, Inc.]
0000 ERRORS [PL1 Rev. 22.0]

OK, BIND
[BIND Rev. 21.0.1 Copyright (c) 1985, Prime Computer, Inc.]
: LOAD UPDATE
: LI PL1LIB
: LI
BIND COMPLETE
: FILE

OK, RESUME UPDATE

second file record
New record:   New second record
OK,
```

DELETING RECORDS

The primary key deletes records from a MIDASPLUS file. The index subfile entry is marked for deletion and the corresponding primary index value is deleted. Remember 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, but a DELETE (with or without a KEY) or a REWRITE operation after a DELETE signals an error.

The DELETE Statement

To delete a record from a MIDASPLUS file, use the DELETE statement. Use DELETE with a KEY option to indicate which record is to be deleted:

```
DELETE FILE(filename) [KEY(keyvar)];
```

If specified, make sure that keyvar is a key value that occurs in the MIDASPLUS file, otherwise a KEY error occurs. If no KEY is specified, the current record is deleted. (It is assumed that a 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.

Delete Program: This program deletes a record from the Sample file:

```
OK, SLIST DELETE.PL1
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 */
dcl sysprint file;

/* KVAR is used with KEYTO option and must be CHAR VARYING */

        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, PL1 DELETE.PL1
[PL1 Rev. T1.0-21.0 Copyright (c) 1986, Prime Computer, Inc.]
0000 ERRORS [PL1 Rev. 22.0]

OK, BIND
[BIND Rev. 21.0.1 Copyright (c) 1985, Prime Computer, Inc.]
: LOAD DELETE
: LI PL1LIB
BIND COMPLETE
: FILE

OK, R DELETE

New second record
KEY(SAMPLE) raised in DELETE at 4673(3)/1224
(record not found in READ)
```

```
ERROR (file = SAMPLE) raised in DELETE at 4673(3)/1224
(no on-unit for KEY)

ER!
```

The error conditions are raised because there is no on-unit to trap key errors. See See Key Errors below.

ACCESSING CREATK-DEFINED FILES

PL/I programs can access MIDASPLUS files that are not created through PL/I. Remember to consider the restrictions on READ and WRITE argument size. It is easier to determine how the arguments should be declared if the file in question has fixed-length records.

Note

Avoid updating the same MIDASPLUS file with more than one high-level language interface.

Reading Fixed-Length Records

When opening an existing CREATK-defined MIDASPLUS file with fixed-length records, determine the record size so that 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.

The data size is displayed next to ENTRY SIZE. (Since PL/I cannot create a MIDASPLUS file with fixed-length records, the ENTRY SIZE is displayed as USER-SUPPLIED for PL/I-created MIDASPLUS files.)

The following program opens and reads the sample MIDASPLUS file BANK, which is a CREATK-defined file:

```
OK, SLIST BANKREAD.PL1
 bankread:
        proc options(main);

/* this program opens and reads from */
/* a previously-created MIDASPLUS file BANK */
/* which has fixed-length records */

dcl bank file keyed sequential;
dcl pkey char(9);
dcl sysprint file;
dcl readvar char(86);
/* 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(bank) begin;
            close file(bank);
            put skip list('End of file');
            stop;
            end;
        open file(bank) input;
        pkey = '189264289';
        read file(bank) into(readvar) key(pkey);
        put skip list(readvar);
        close file(bank);
        end;

OK, PL1 BANKREAD.PL1
[PL1 Rev. T1.0-21.0 Copyright (c) 1986, Prime Computer, Inc.]
0000 ERRORS [PL1 Rev. 22.0]

OK, BIND
[BIND Rev. 21.0.1 Copyright (c) 1985, Prime Computer, INC.]
: load bankread
: li pl1lib
: li
BIND COMPLETE
: file

OK, R BANKREAD

189264289murray, paul              mc28374646123 orchard rd  manchester     nh03102
OK,
```

ERROR HANDLING

The most commonly encountered errors in the PL/I interface are key and record errors. 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 error 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. (The default for PL/I-created files is 32 characters.)

- There is no more room to add keys in the primary index subfile. (This error is very rare.)

- 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.)

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:

```
OK, SLIST READERR.PL1
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;
dcl sysprint file;

/* 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;
        /* now run the program */

OK, PL1 READERR.PL1
[PL1 Rev. T1.0-21.0 Copyright (c) 1986, Prime Computer, Inc.]
0000 ERRORS [PL1 Rev. 22.0]

OK, BIND
[BIND Rev. 21.0.1 Copyright (c) 1985, Prime Computer, Inc.]
: LOAD READERR
: LI PL1LIB
: LI
BIND COMPLETE
: FILE

OK, R READERR

first file record
first file record
OK,
```

Record Errors

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 requires that the record size and the read variable match exactly. Careful programming is the only way to avoid this error in PL/I.

Other Possible Error Conditions

The undefined file condition may be raised during an unsuccessful attempt to open a file. This can occur for a variety of reasons:

- The filename was misspelled when you opened the file for INPUT or UPDATE. (Use of these access modes assumes that the file exists.)
- An attempt was made to open a non-existent file for INPUT or UPDATE.
- Incomplete or conflicting attributes were specified during an OPEN or DECLARE FILE statement.
- Your version of PL/I has not been loaded with the proper MIDASPLUS interface routines and/or proper libraries.
- Your compiler is not functioning properly.

In MIDASPLUS files, if one of the segment subfiles is left open (due to program error or failure) an attempt to re-open the file will cause this condition. Use the PRIMOS level command CLOSE ALL to close the file before attempting to rerun the program.

Locked Records

If the program aborts and a record in the file remains locked, run MPACK to unlock the record since the record cannot unlock itself. If a record is locked and you press CTRL-P or BREAK immediately, the record stays locked. If you attempt to read a locked record, an error condition is signaled and the program fails. At this point, run the MPACK utility using the UNLOCK option.

9

The VRPG Interface

This chapter explains how to use VRPG to access MIDASPLUS files. The chapter discusses the language dependent features of VRPG, describes a MIDASPLUS file in VRPG, explains the file operations, provides VRPG programming examples, explains multiple key processing, and describes IBM system/34 compatibility features. The following is an overview of the VRPG Specification Statements. (This chapter refers to the standard VRPG coding Specifications Sheet as a statement.)

File Specification describes the name of a file and its attributes.

Input describes all input record descriptions for input or update files.

Calculation describes calculations performed on data. Allows the programmer more flexibility beyond sequential processing.

Output describes all output records for output or update files.

Extension describes the tables and arrays.

Prime's VRPG supports both keyed-indexed and direct access MIDASPLUS files. You describe the file organization in the File Specification Statement.

LANGUAGE-DEPENDENT FEATURES

VRPG can use the primary key and up to 17 secondary keys to access a MIDASPLUS file. For information about using VRPG with secondary keys, see the section MULTIPLE KEY PROCESSING later in this chapter. Special considerations for the VRPG interface with MIDASPLUS follow:

- All keys must be in the data record.
- VRPG does not support secondary data.
- All keys of an indexed file can be no more than 32 characters long.
- During template creation with CREATK, the A (ASCII) option defines a primary key of a direct file. This primary key, defined as a standard numeric item, is called the relative record number.
- If the MIDASPLUS files are indexed and are designated as chained files, you can use a key to randomly access the files.
- If the MIDASPLUS files are direct files and are designated as chained files, you can use a relative record number to randomly access the files.
- You can only delete records from indexed files.

COMPILE AND LOAD SEQUENCE

Substitute the name of your program for the word program in the example. A sample compiling and linking session using VRPG follows.

```
vrpg program
[VRPG Rev. 19.4]
F
I
O
0000 ERRORS [VRPG Rev. 19.4]
OK, bind
[BIND rev 19.4]
: load program
: li vrpglb
: li
BIND COMPLETE
: file
OK, resume program
```

DESCRIBING A MIDASPLUS FILE IN VRPG

The first part of MIDASPLUS file handling in VRPG is the definition process. This process consists of correctly describing the MIDASPLUS file to VRPG via the File Specifications Statement. Table 9-1 shows how to define a keyed-index or a direct access MIDASPLUS file with the File Specifications Statement. Table 9-2 lists the MIDASPLUS specific fields in the other VRPG statements. See the RPG II V-Mode Compiler Reference Guide for complete details of these statements.

Table 9-1
VRPG File Description Specifications
For MIDASPLUS Files

Attributes	Column(s)	What to Specify
File Type	15	I = Input O = Output U = Update
File Designation	16	P = Primary S = Secondary C = Chained D = Demand (Blank = Output)
File Format	19	F = Fixed-length
Record Length	24-27	MIDASPLUS data record length, including the Primary key length.
Mode of Processing	28	R = Random by key, relative record number, or ADDROUT file. L = Sequential within limits, or by record address file. Blank = Sequential by key (indexed files) or consecutive access (direct files).
Key-Field Length	29-30	Number of characters in index -- for indexed files only. (Length from 1-32)
File Organization	32	I = Indexed D = Direct

Table 9-1 (continued)
VRPG File Description Specifications
for MIDASPLUS Files

Attributes	Column(s)	What to Specify
Key Col. Position	35-38	Column where the key starts in the data record -- for indexed files only.
Extension Code	39	M = Multiple keys are being used. Key definition lines follow that give all of the keys used in the indexed file. For information about key definition lines, see the section MULTIPLE KEY PROCESSING later in this chapter.
Device	40-46	DISK (Required)
File Addition	66	A = Add records to a non-empty indexed file. The records do not have to be added in sequential order. U = Add unordered records to an empty indexed file. Must be used in conjunction with an O in column 15. Blank = Add ordered records to an empty indexed file. Must be used in conjunction with an O in column 15.
Key of Reference	73-74	Number (00-17) indicating the key being defined in columns 35-38 -- for indexed files only.

A few of the attributes mentioned in Table 9-1 require more explanation regarding their relationship to MIDASPLUS. This section describes how file type specification (column 15), file designation restrictions (column 16), file addition specifications (column 66), and mode of processing (column 28) relate to MIDASPLUS file processing in VRPG.

Table 9-2
VRPG Fields For Other Statements

Statement	Column(s)	What to Specify
Calculation	28-32	Operation: SETLL, SETK, READ, CHAIN
Calculation	54-55	Indicator on for record not found
Extension	11-18	Name of the separate sequential limits file for the RAF method
Extension	19-26	Name of the MIDASPLUS file to be processed
Input	61-62	Matching fields or chaining fields
Output Specifications	16-18	ADD = Add records to an indexed file DEL = Delete records (Blank = Add records to a direct file)

File Type Specification

The following restrictions apply to MIDASPLUS 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, writing, deleting, and updating) *

* If an A occurs in column 66 of the File Description Statement and an ADD entry appears in column 16-18 of the Output Statement, new records may be added to the file.

You cannot describe a MIDASPLUS file as a Display (D) file in VRPG.

File Designation Restrictions

The legal file designations for a MIDASPLUS file, as indicated in column 16 are:

- Primary (P): The main file from which records are read. Specify only one primary file per program. You can open the primary file for input or update.

- Secondary (S): One or more files from which records are read after the Primary file is processed. This happens if matching is not specified in columns 61-62 of the Input Statement. If matching is specified, standard VRPG matching algorithms process secondary files. Secondary files can be Input or Update files, and they are processed in the order in which they appear in the File Description Statements.

- Chained (C): Either read randomly or loaded directly via the CHAIN operation code in the Calculation Statement. Chained files can be opened for Input, Output, or Update.

- Demand (D): Can be either 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

You can add records either sequentially or randomly to an indexed (MIDASPLUS) file. Records cannot be added to a file opened for access under the sequential within limits processing mode. For an indexed file opened for Input, Output, or Update, A in column 66 indicates that 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 you are required to supply records in key order. 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) since MIDASPLUS does not have to sort the records in the loading process.

To initially load records to direct files, specify the file as Output Chained. Put O (Output) in column 15, and C (Chained) in column 16. To add records to a direct file that already contains entries, specify the file as Update Chained. Put U in column 15 and C in column 16. In either case, column 66 of the File Specification Statement and columns 16-18 of the Output Statement should be blank. The relative record number values cannot be larger than the number of records pre-allocated during template creation.

Primary or Secondary Files:

You may access primary or secondary files by sequential or random access methods. The particular access method (mode of processing) depends on the file's organization.

Organization	Mode of Processing
Direct	Sequential by relative record number, Random by ADDROUT file
Indexed	Sequential by key, Sequential within limits, Random by ADDROUT file

Chained Files

You may process MIDASPLUS files declared as Chained files according to their organization. If the MIDASPLUS file is indexed, access it randomly by key. If the file is direct, access it randomly by relative record number only. You can only access MIDASPLUS files by key if they are defined as Chained files.

Demand Files

You may process demand files (both indexed and direct) sequentially by key or sequentially within limits (for indexed files only). There are two methods of accomplishing limits processing: using the set lower limit (SETLL) operation in a Calculation Statement, or using a record address file (RAF). (RAFs are sequential files used to perform limits processing on indexed files.)

The SETLL operation specifies a lower limit for the primary key value. The file can then be positioned to the record having that primary key value and can be processed from that point.

The RAF method requires the creation of a separate sequential "limits" file. The file contains records that specify the low value from which to start processing, and the high value at which to stop processing. To use the RAF method, specify the name of this file in the Extensions Statement (in columns 11-18), along with the name of the MIDASPLUS file that is to be processed (columns 19-26).

FILE OPERATIONS

The operations that can be performed on a MIDASPLUS file vary with the file description in the File Description Statement. The following are the standard file operations that a VRPG program can perform on MIDASPLUS files:

- Position the file
- Read records
- Add records
- Load records
- Update records
- Delete Records

These operations are explained below.

Positioning the File

File position is thought of in terms of a file pointer that 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 can be positioned to a specific record in the file. Use a CHAIN operation in the Calculation Statement to establish file position. When you specify a primary key value for a chained file, the file pointer positions to the record with this key value. This record becomes the current record and is read. If the file is opened for Update, the record will also be locked.

Demand files can also be positioned, but only indirectly. SETLL or a record address file (RAF) can do the positioning. Set the lower limit for the primary key with the SETLL operation in a Calculation Statement or in the RAF, and then perform a READ operation. The record positioned to will be the one whose primary key value matches the lower limit value, if such a record exists. If this primary key value does not exist, the next record whose primary key value is greater than the lower limit specification becomes the current record.

Reading Records

Depending upon their designation, you can read MIDASPLUS files as part of the normal Input cycle or as part of the Calculation cycle. 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 RAF to set an initial file position. The

read occurs during the READ operation of the Calculation Statement. Using the CHAIN operation of the Calculation Statement, the primary or relative record key value randomly reads records from files declared as chained. If the file is an Update file, the record is also 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 the order in which the key values appear in the primary index subfile. After a record is read, VRPG automatically advances the file pointer and makes 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. You cannot randomly read files opened as primary or secondary files.

Demand files can be read sequentially within limits using the SETLL or RAF operations.

Random Reads: Random reads can be performed only on files designated as chained files. This applies both to indexed and direct files. To perform random reads, use the CHAIN operation in the Calculation Statement. The primary key or relative record number is supplied to MIDASPLUS, and the next record whose primary key matches this value is returned. Set the indicator for "no record found" (in columns 54 and 55 of the Calculation Statement). This allows 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 during template creation. If a legal record number is supplied in a CHAIN operation, but there is no corresponding record for that number, the record is returned as blanks. VRPG 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.

Adding Records

Use standard VRPG output methods or the KBUILD utility to add records to a MIDASPLUS file.

In indexed files that contain entries, if column 66 of the File Specifications Statement contains an A and columns 16-18 of the Output Statement specify the ADD Operation, VRPG can add records to the file. The file can be opened for Input, Output, or Update. This applies only to files that already contain entries.

Direct access files do not support ADD because of their file structure. Use an update to add records. (See Updating Records below.)

Loading Records

To load an indexed file, place a U or a blank in column 66 of the File Specifications Statement. U implies an unordered load and blank implies an ordered (sequential by key) load. 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. 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 MIDASPLUS file record in VRPG, you must first read the record and then update it in the Output cycle. Be careful not to change the primary key value when rewriting a record. MIDASPLUS 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. VRPG assumes that any record for which space has been pre-allocated by CREATK, but that was not added during the initial load, exists as all blanks. As a result, any attempt to add a new entry to a direct access file is an update of the blanks that already appear in this slot in the file.

Deleting Records

VRPG allows you to delete records for indexed files. In order to delete a record, either use a CHAIN operation in the Calculation Statement to read the file or use READ statements to read the file and then use an output specification to delete the records. To delete records, you must specify a file as an update file. The record that was previously read is the record that will be deleted.

Use header, detail, total, or exception output in the output specifications to specify deletion. Specify DEL in columns 16-18 of the main line in an output specification along with the filename, the type of output, and any output indicators. You may use OR-lines to condition the deletion, but DEL should only be specified in the first line of the specification. DEL applies to all of the OR-lines. Do not give any field specifications for a deletion.

When a delete occurs, the key and record are deleted from the file and are no longer available. MIDASPLUS marks the record as deleted and

physically removes the record if MPACK is used. (See Chapter 15, PACKING A MIDASPLUS FILE.) The following compilation errors and their meanings can occur with DEL:

Severity 3 error	DEL is specified for a file that is not an update indexed file.
Severity 2 error	An output field is specified for a DEL output. All specifications are ignored.
	DEL is specified on an OR-line, but was not specified in the main line. DEL is ignored.

If DEL is specified on an OR-line and was specified in the main line, no error occurs. The program is treated as if there were no DEL on the OR-line.

Error Handling

A runtime error occurs if a program tries to delete a record without successfully reading it first. You may enter S and the program continues, ignoring this deletion attempt. The filename and the last key processed (if a key existed) are displayed with the error message. Runtime errors also occur if MIDASPLUS is unable to delete the record.

VRPG flags all MIDASPLUS errors but can only recover from "record locked", "record not on file", and "delete before a successful read" conditions. The messages are described below. The format of the messages is shown here.

Record Locked: The record locked message is:

```
**** ERROR ****
Unable to lock the record for update. The record is already locked.
File :  filename
Key :  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 MIDASPLUS 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 enter S, the operation is executed again.

CHAIN Errors: An unsuccessful CHAIN operation invokes the message:

```
UNSUCCESSFUL CHAIN.  TYPE S(CR) TO SKIP PAST OUTPUT.
```

This message is returned when a CHAIN operation has been attempted on the file and MIDASPLUS 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 MIDASPLUS file. Entering S restarts the VRPG program cycle and skips all of 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. You have no control over such errors.

The general MIDASPLUS error message is reported as:

```
MIDASPLUS [MIDASPLUS error message number] filename
```

MIDASPLUS errors cannot be handled by typing an S followed by RETURN. Type an S only when VRPG's message states to do so.

MIDASPLUS Concurrency Errors: VRPG returns the MIDASPLUS concurrency error message as:

```
**** ERROR ****
A concurrency error occurred while updating the record.
Another program probably added a record into this position
in between the read and the output.
File : filename
Record : record
Key : key
PROGRAM EXECUTION TERMINATED. (VRPG)
```

Key represents the key value that VRPG was processing when the error occurred.

In a multiuser environment, it is possible for several users 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 cannot happen with two VRPG users, it is possible that some FORTRAN or COBOL user may have the same file open for update while a VRPG user is simply reading from it. The VRPG user may get the above message when attempting to update a now-deleted record or when the file has been changed because of updated or added records.

INDEXED FILE EXAMPLES

The first example shows how a MIDASPLUS template can be created, populated, and accessed through VRPG. The second example shows how to delete indexed MIDASPLUS file records.

Example 1:

The indexed file Master is created using CREATK as shown in this sample session. The primary key is defined as an ASCII key of 5 characters in length. The data size is defined as 32 words (64 characters).

```
OK, creatk
[CREATK rev 19.4.0]

MINIMUM OPTIONS? yes

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

DATA SUBFILE QUESTIONS

PRIMARY KEY TYPE: a
PRIMARY KEY SIZE = : b 5
DATA SIZE IN WORDS = : 32

SECONDARY INDEX (CR)

INDEX NO.? (CR)

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

Although the above example shows how to create a template with only one key, you can create a file with multiple keys, and use those keys to access the file. For more information, see the section MULTIPLE KEY PROCESSING later in this chapter.

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.

The H that appears at the top of the program is a Header card. It 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                        U
ISFILE   NS  01
I                                          2   60EMPNO
I                                          7  64 DATA
OMASTER  D        01
O                                      1 'P'
O                        EMPNO         6
O                        DATA         64
 *
```

The sequential file SFILE contains the following records:

```
04416124 ADAMS  WJ01234567800MH  200 380000  32000
22236124 BROWN  HY23456789000MH20000 800000  60000
25781125 COOPER IG333445555O2SH  450 750000  52000
39840124 DAVIS  TV44455666601MH  250 400000  30000
47124123 EVANS  AS345678901OOMH350001400000 120000
66031123 FOX    FL456789012OOMH  350 660000  26000
73315125 HOLMES EB56789012300MH10000 400000  30000
80081125 JONES  COOOO11222200SH  400 640000  64000
86789123 KELLER ND999887777O1MS  300 520000  38000
98570124 LAKE   MP888776666O4SS300001200000  80000
```

When the program above is run, the records in the sequential file are added to the MIDAPLUS file Master. To read the records back from the file and print out a report showing what is in the file, use the READ program, listed below. The READ 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   6OEMPNO
I                                          7   9 DEPT
I                                         10  16 NAME
I                                         17  18 INIT
I                                         19  27 SSN
I                                         28  29OEXEM
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
  *
```

```
OK, vrpg read
[VRPG Rev. 19.4]
F
I
C
O
0000 ERRORS [VRPG Rev. 19.4]
OK, bind
[BIND rev 19.4.1]
: load read
: li vrpglb
: li
BIND COMPLETE
: file
OK, resume read
OK, slist PRINT
```

```
 5/07/85                          EMPLOYEE LISTING                          PAGE    1
EMPLOYEE   EMPLOYEE     DEPARTMENT    RATE       Y-T-D        Y-T-D        Y-T-D
 NUMBER      NAME         NUMBER                  GROSS         TAX          NET

  00811   JONES   CO        25         4.00     6,400.00      640.00     5,760.00
  22361   BROWN   HY        24       200.00     8,000.00      600.00     7,400.00
  33151   HOLMES  EB        25       100.00     4,000.00      300.00     3,700.00
  44161   ADAMS   WJ        24         2.00     3,800.00      320.00     3,480.00
  57811   COOPER  IG        25         4.50     7,500.00      520.00     6,980.00
  60311   FOX     FL        23         3.50     6,600.00      260.00     6,340.00
  67891   KELLER  ND        23         3.00     5,200.00      380.00     4,820.00
  71241   EVANS   AS        23       350.00    14,000.00    1,200.00    12,800.00
  85701   LAKE    MP        24       300.00    12,000.00      800.00    11,200.00
  98401   DAVIS   TV        24         2.50     4,000.00      300.00     3,700.00

TOTAL SALARIED EMPLOYEES:  2                   71,500.00    5,320.00    66,180.00
TOTAL HOURLY EMPLOYEES:    8
```

Example 2:

The indexed file INP2 is created using CREATK as shown in this sample session. The primary key is defined as an ASCII key of 2 characters in length. The data size is defined as 40 words (80 characters).

```
CREATK
[CREATK rev 19.4.0]

MINIMUM OPTIONS? yes

FILE NAME? inp2
NEW FILE? yes
DIRECT ACCESS? no

DATA SUBFILE QUESTIONS

PRIMARY KEY TYPE: a
PRIMARY KEY SIZE = : b 2
DATA SIZE IN WORDS = : 40

SECONDARY INDEX (CR)

INDEX NO.? (CR)

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

The following KBUILD session populates the file.

```
OK, kbuild
[KBUILD rev 19.4.0]

SECONDARIES ONLY? no
ENTER INPUT FILENAME: data
ENTER INPUT RECORD LENGTH (WORDS): 40
INPUT FILE TYPE: text
ENTER NUMBER OF INPUT FILES: 1
ENTER OUTPUT FILENAME: inp2
ENTER STARTING CHARACTER POSITION, PRIMARY KEY: 1
SECONDARY KEY NUMBER: (CR)
IS FILE SORTED? no
ENTER LOG/ERROR FILE NAME: (CR)
ENTER MILESTONE COUNT: 100

BUILDING: DATA
DEFERRING: 0
```

```
PROCESSING FROM:  data
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM     DIFF
     0  05-07-85  10:54:20       0.000      0.000      0.000      0.000
FIRST BUILD/DEFER PASS COMPLETE.
    11  05-07-85  10:54:20       0.003      0.000      0.003      0.003

SORTING INDEX 0
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM     DIFF
     0  05-07-85  10:54:20       0.000      0.000      0.000      0.000
SORT COMPLETE
    11  05-07-85  10:54:20       0.005      0.000      0.005      0.005

BUILDING INDEX 0
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM     DIFF
     0  05-07-85  10:54:20       0.000      0.000      0.000      0.000
INDEX 0 BUILT
    11  05-07-85  10:54:20       0.002      0.000      0.002      0.002

KBUILD COMPLETE.

OK,
```

The input file DATA contains the following records:

```
01**FIRST RECORD**
02**SECOND RECORD**
03**THIRD RECORD**
04**FOURTH RECORD**
05**FIFTH RECORD**
06**SIXTH RECORD**
07**SEVENTH RECORD**
08**EIGHTH RECORD**
09**NINTH RECORD**
10**TENTH RECORD**
99**LAST RECORD**
```

The input file INP1 contains the following records:

```
03
01
66
07
03
05
35
09
01
99
```

The following program, DEL1, loads a sequential disk file called INP2.

```
     F*    DEL1
     F*    This program tests the deletion of records from
     F*    an indexed file using CHAIN reads to the indexed file.
     F*
     F*    The program gets the keys from the sequential file INP1.
     F*    If the key matches with a record in the indexed file, the
     F*    record is deleted.  The output file records each key read
     F*    from the sequential file and the record of the indexed file,
     F*    if the key is in the indexed file.
     F*
     FINP1    IP  F  80  80           DISK
     FINP2    UC  F  80  80R 2AI    1 DISK
     FOUT1    O   F  80  80           DISK
     IINP1    AA  01
     I                                        1   20FIELD1
     IINP2    BB  02
     I                                        1  30 FIELD2
     C           FIELD1    CHAININP2                 91
     C                     SETOF                     03
     C  N91                SETON                     03
     OINP2    DDEL      03
     OOUT1    D         01
     O                                  20 'FIELD: '
     O                          FIELD1  40
     O                    03    FIELD2  80
/*
OK, vrpg dell
[VRPG Rev. 19.4]
F
I
C
O
0000 ERRORS [VRPG Rev. 19.4]
OK, bind
[BIND rev 19.4.1]
: load dell
: li vrpglb
: li
BIND COMPLETE
: file
OK, resume dell
```

```
OK, slist out1
          FIELD:          03          03**THIRD RECORD**
          FIELD:          01          01**FIRST RECORD**
          FIELD:          66
          FIELD:          07          07**SEVENTH RECORD**
          FIELD:          03
          FIELD:          05          05**FIFTH RECORD**
          FIELD:          35
          FIELD:          09          09**NINTH RECORD**
          FIELD:          01
          FIELD:          99          99**LAST RECORD**
OK,
```

DIRECT ACCESS IN VRPG

VRPG supports direct access MIDASPLUS files as standard VRPG direct files. When a direct access MIDASPLUS file template is created specifically for use with VRPG, define the primary key as an A (ASCII). In VRPG programs that process direct access files, specify the file organization as direct (D) and open the file as a Chained file. Records are read randomly by record number.

Updating Records

In Update mode, you can add new records to a file (if it already contains data) and rewrite existing data records. To update a direct file, use a sequential file that tells the update program what to do with certain records in the direct file.

MULTIPLE KEY PROCESSING

VRPG allows for MIDASPLUS indexed file processing using primary and secondary keys and supports the use of partial keys to read a record. This section contains the details of defining an indexed file and its keys, as well as reading, writing, updating and deleting records. Most of this information is basic to primary key as well as secondary key access. Although taking advantage of the new functionality requires new programming techniques, existing VRPG programs do not need to be altered in order to function as they always have.

Defining the File and the Keys with CREATK

Since VRPG cannot create a MIDASPLUS file template from program level, you must create all MIDASPLUS files with the utility CREATK. By using the CREATK dialog, you can define all primary and secondary keys for MIDASPLUS indexed files.

Use the following guidelines when creating a MIDASPLUS keyed-index file template with secondary keys:

- Up to 17 secondary keys are allowed.
- Duplicate secondary keys are allowed.
- All keys must be contained in the data record.
- Although keys may be anywhere in the data record, it is recommended that they not overlap the primary key field.
- Keys may be of type ASCII or type bit string, and have a maximum length of 32 characters; packed, binary, floating point, or concatenated keys are not allowed.
- Secondary data is not supported.

A sample CREATK dialog for an indexed file template with two secondary keys follows. Note that the dialog requests some sizes be specified in words. This refers to 16-bit entities or half-words.

```
[CREATK Rev. 20.0 Copyright (c) Prime Computer, Inc. 1985]

MINIMUM OPTIONS? YES

FILE NAME? TESTFILE
NEW FILE? YES
direct ACCESS? NO

DATA SUBFILE QUESTIONS

PRIMARY KEY TYPE: A
PRIMARY KEY SIZE = : B 8
DATA SIZE IN WORDS = : 64

SECONDARY INDEX

INDEX NO.? 1

DUPLICATE KEYS PERMITTED? NO

KEY TYPE: A
KEY SIZE = : B 32
SECONDARY DATA SIZE IN WORDS = : <CR>

INDEX NO.? 2

DUPLICATE KEYS PERMITTED? YES
```

```
KEY TYPE: A
KEY SIZE = : B 8
SECONDARY DATA SIZE IN WORDS = : <CR>

INDEX NO.? <CR>

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

In the above example the key type is specified as ASCII, and the key size is given in bytes. When the key is ASCII, the key size can also be specified in half-words. Bit string keys are also allowed, and the size then is given in bits (multiples of 8 only) or half-words. It is usually easier to define keys as ASCII and use bytes for key size, since these specifications correspond to the key definition in a VRPG program.

File Specification for Multiple Key Processing

In order to use multiple keys to access a MIDASPLUS file, you must make the appropriate definitions in the file specification section of your program. The new elements of the F-Spec lines that are common to all multiple key processing are discussed below.

The Main File Specification Statement: As in the past, columns 35-38 of the main File Specification Statement are used to define a key, but this key can now be any one of the keys of the MIDASPLUS file. The key defined on the main File Specification Statement is called the beginning key of reference because it is the initial key that the program uses to reference the data entries. You may change the key of reference later in the program.

Format the main File Specification Statement as you always have, with the following exceptions:

Column 35-38:	The beginning key of reference's starting position in the data record. The beginning key of reference may be the primary key or any existing secondary key.
Column 39:	An M, to indicate that multiple keys are being used.
Columns 73-74:	The key index number (00-17) of the beginning key of reference. The index number must be the same number that was assigned to the key during template creation.

It is not absolutely neccessary to indicate an index number in columns 73-74. If you do not indicate an index number, the compiler searches the continuation lines (explained below) for a key definition that matches the one on the main File Specification Statement, and makes that key the beginning key of reference. However, identifying the key index number in the main File Specification Statement helps the compiler to run more efficiently and serves to better document your program.

Continuation Lines: All of the MIDASPLUS file's keys, including the key defined in the main File Specification Statement, must be defined on continuation lines, one line for each key. For an input file, you only need to define the keys that you will be using to access the data records. For update or output files, define all of the file's keys.

The definitions that the continuation lines provide are used at runtime to tell MIDASPLUS which key indexes to update. If you do not define a key, it will not be updated in the key index, even though you may have changed the key field in the data record. An undefined key that has been changed in the data record does not cause the program to abort, nor does it cause an error message. An undefined key causes you to end up with a file whose key indexes do not match the corresponding fields in the data record. To avoid this situation, define all of the keys of your MIDASPLUS file whether you use the file for Input, Output, or Update.

The File Specification Statement continuation lines supply the following information:

- Key number (00-17)
- Key's length (1-32 characters)
- Key's starting position in the data record
- Indication whether or not duplicates are allowed for a secondary key

The key number on the key definition lines must correspond to the numbers used when the file is defined with CREATK. You must structure the continuation lines for key definition in the following format:

Column 6: F (File specification)

Columns 26-27: A key number (0-17), right justified

Columns 29-30: Length of the key (1-32), right justified

Columns 35-38: Key start location in the record, right justified

Column 39: D, if duplicates are allowed (secondary key only)

Column 67: K (indicates key definition line)

The primary key is always number 0, and secondary keys are numbered 01 through 17. The key index number must match the number assigned to it during template creation. If the duplicate field is blank, duplicates are not allowed.

The following is an example of the file specification lines that define the indexed file that was created above with CREATK. The file is an Update Chained file. Note that a key number is not specified in columns 73-74 of the main File Specification Statement line. However, the key defined on this line has the same attributes as the first secondary key. The compiler, therefore, assigns the first secondary key to be the beginning key of reference. As stated before, an entry in columns 73-74 is not absolutely neccessary, but the program compiles more efficiently if one is included.

```
FTESTFILEUC  F     128R32 I     9MDISK
F*                               |
F*                               Indicates Multiple keys.
F*                               Following F-lines define keys.
F*
F* Primary Key:
F                   00  8       1                              K
F* 1st Secondary Key:
F                   01 32       9                              K
F* 2nd Secondary Key:
F                   02  8      41D                             K
```

Changing the Key of Reference in a Program

Once you have defined the keys of the MIDASPLUS file in the File Specification Statements, you may use the new operation code opcode, SETK, to change the key of reference. The format of the SETK statement is:

Columns 18-27 (factor1): a key number, left justified

Columns 28-32 (opcode): SETK

Columns 33-42 (factor2): the Indexed filename

The SETK opcode may be conditioned by indicators in columns 9-17, but may not have resulting indicators in columns 54-59. Any entries in other columns of the SETK statement are ignored. If the key number was not used previously to define a key definition line, a compiler error occurs. The SETK statement is used in the Calculation Statement before a SETLL, READ, or CHAIN statement when the program wishes to reset to a new key of reference.

Single Key Processing of a Multiple Key File

With an input file, you do not need to define all of the keys in order to access a data record. If you plan to access a file using only one key of reference, you can omit the continuation lines that define keys and define your key of reference in the main File Specification Statement. This is true whether the key of reference is the primary key or a secondary key. For single key access, you must always specify the key of reference's index number in columns 73-74, unless the key is the primary key. The main File Specification Statement is formatted as usual except that you:

- Indicate the key of reference's starting position in the data record in columns 35-38
- Indicate the number of the key (00-17) in columns 73-74

When a key of reference is defined in the above manner, you will not be able to change it within a program.

This new functionality does not require any changes to your old VRPG programs if you want to run them as in the past, namely, with the primary key as the sole key of reference. Similarly, if you want to write new programs that do not use the new functionality, you can program in the same fashion as you always have, ignoring columns 73-74 of the main File Specification Statement and the continuation lines that define secondary keys.

PROCESSING WITH SECONDARY KEYS

The following discusses the basics of multiple key file access. All basic types of indexed file access are included, with specifics noted for secondary key access.

Position File by Secondary Key: Indexed files may be processed sequentially within limits. In the main File Specification Statement, describe the file as Input or Update, and Primary, Secondary or Demand. Put an L in Column 28 of the File Specification Statement to specify limits processing.

Use the SETLL opcode in a Calculation Statement to set the lower limit. Specify a data name or constant in factor1 (columns 18-27) of the SETLL calculation statement. Specify the name of the file to be processed in factor2 (columns 33-42).

Specify the key of reference in the main File Specification Statement, or use a SETK statement before the SETLL statement.

Read Next Record by Secondary Key: Indexed files can be processed sequentially by secondary key. This processing can occur either by a demand read or in the normal cycle.

To process a Demand file sequentially, use the READ opcode in a calculation statement. Leave factor1 of the calculation statement blank. (The current key of reference is used for the read.) Specify the name of the indexed file to be read sequentially in factor2. This type of operation can be done with or without limits processing.

If the file is not specified as a Demand file, sequential input is done through the normal program cycle, using the current key of reference.

With either type of sequential read, the key of reference can be altered by a SETK statement. The reads continue at the first key value in the key subfile indicated by the SETK statement. In the normal program cycle, an indicator should condition SETK. You can then set this indicator off after the SETK, avoiding the possibility of an infinite loop.

Read Record Randomly by Secondary Key: You can specify a key value to process indexed files randomly. Describe the file as either Input or Update, and Chained. Put an R in column 28 of the File Specification Statement to indicate random processing.

Use the CHAIN opcode in a calculation statement. Specify a data name containing the value of the key, or a constant, in factor1. Make sure that the proper key of reference is set. Specify the name of the file in factor2.

Random and Sequential Processing with the Same File: VRPG runtime was modified to allow random and sequential processing with the same file. This type of processing is useful for accessing all the records in a file that have duplicate key values.

Define the file as Chained in the main File Specification Statement. Put an R in column 28.

Use a CHAIN statement to set the file pointer to the first occurrence of the key value. The Random read is accomplished by a CHAIN statement, setting the file pointer to the appropriate record. Then, use a standard READ statement to move through the file sequentially from the current position. Normally, the initial READ operation starts at the first record in the current index, and then reads sequentially from there. The record following the current record is returned only when a READ statement immediately follows a CHAIN statement.

When processing a file with duplicate secondary keys, code the program to check that the record sought by a CHAIN or READ command was found. If a secondary index has duplicate values, a CHAIN statement will always return the first record in the series of duplicate key values. Use a READ statement to get to the duplicates.

Updating the Current Record: Update a current record during output in the normal program cycle, or by exception output (using EXCPT in a Calculation Statement).

Define the file as Update. The Update operation must be preceded by a successful READ operation (either CHAIN or READ) in the same program cycle as the Output/Update operation. When an update occurs, all secondary key fields in that record that have been modified will have their corresponding key index entries updated.

Define any keys that you want updated on key definition lines or else the MIDASPLUS key index will not be updated. The update of the record occurs even if the key field is not specified on the output record. Update files with secondary keys must always have a primary key definition line included, or a compiler error occurs.

VRPG checks to see if you have inadvertently created duplicates in a secondary key where they are not allowed. If this condition occurs, VRPG issues a runtime error message and halts execution if one is found. In such cases, the record and any of its key index entries is not updated. All data in the MIDASPLUS file is returned to the state that it was in prior to the update attempt.

If the key field in the record that is being updated is also the key of reference, the same record might be read again later in the sequence. Therefore, it is recommended that you do not update the key of reference in a file that also is being read sequentially.

Also, if a key field being updated overlaps another key field, the key index entry might not be what is expected. VRPG runtime performs the secondary key index update based upon the key's value in the constructed output record. The key may not even have been specified as an output field, but the key index update will still occur if its value in the record has changed.

Write Record and Keys: This operation is usually referred to as loading the file, as the file is empty at the beginning of the WRITE operation. Loading a file is handled during output during the normal program cycle or by exception output (EXCPT).

Describe the file as Output in the main File Specification Statement. The file may be loaded in ordered or unordered sequence. To specify an unordered load, put a U in column 66 of the main File Specification Statement; for an ordered load, column 66 is blank.

During an ordered load, the given key of reference value for each record is checked against the key's value in the previous record. The keys must be in ascending ASCII sequence. If a record is encountered that is out of sequence, a runtime error occurs and the record is not added to the file. Do not change the key of reference in an ordered load program.

At output time all of the MIDASPLUS key indexes (primary and secondary) whose keys have been defined in the File Specifications Statements are updated to reflect the current record. This occurs whether or not the key fields are specified on the Output Specification Statement.

When writing records, VRPG checks for duplicates in the secondary key fields and issues a runtime error if a duplicate occurs where it is not allowed. In such cases, neither of the record's key index fields nor the record itself is added to the file. You are given the option of skipping past the record or terminating execution.

MIDASPLUS always adds index entries in a logically sorted order, independent of the order in which the records are added to the file.

Add Record and Keys: This operation is handled during output in the normal program cycle or by exception output (EXCPT). Put ADD in columns 16-18 of the Output Specification Statement. Put an A in column 66 of the File Specification Statement for the file to which records or keys are being added.

Adding a record and keys differs from writing a record and keys in that the file is usually described as Input or Update instead of Output, and the file is normally not empty. The added records are not checked for any particular sequence. At output time, all keys that have been defined in File Specification Statements are added to the appropriate indexes. Illegal duplicates during an add operation are handled as they are in a WRITE operation.

A VRPG program cannot add a secondary index (define a new secondary index) entry for an existing record.

Deleting a Record and Keys: This operation is handled during output in the normal program cycle. Enter DEL in columns 16-18 of the Output Specification Statement. Define the file as Update. Update files with secondary keys must always have the primary key included in the key definition lines.

A successful read of the record must occur in the same program cycle before the delete. Any key may be used to do the read. The primary key index entry and the record are deleted. The secondary key indexes are flagged by MIDASPLUS for deletion. The actual deletion of the secondary indexes occurs when the key next is accessed, the file is read sequentially, or MPACK is run on the file. VRPG does not allow deletion of a secondary index entry alone.

Partial Key Searches: You may specify a constant or data item with a length less than or equal to the actual key length in factor1 of the SETLL or CHAIN operations. The key is treated as a left-justified partial key. The SETLL statement positions the file at the first record that has a key equal to or greater than the specified partial key value. Similarly, the CHAIN statement retrieves the first record that satisfies the partial key value.

Error Recovery

The VRPG compiler checks the syntax and semantics of secondary key definitions on the File Specification Statements. It is up to you, however, to make sure that the program's key definitions match those of the template. If you are not sure of how the template is structured, you can find out by using the PRINT function of CREATK.

When the file is opened, the VRPG runtime system checks for correct file and key definitions by means of the MIDASPLUS routine KX$RFC. If there is a discrepancy between the program's definitions and the template, the file is not opened, and execution halts.

Always define secondary keys that perform add or delete functions. Any program that is designed to add, delete, or update records must include all of the affected keys to perform these operations correctly.

During runtime, the VRPG library handles errors in the same manner as in the past. This involves issuing a message to the terminal, and in selected cases, giving the user the option to skip the record and continue processing. Other errors cause program execution to halt with proper cleanup procedures (all files closed, and so forth). In some cases, VRPG runtime relies on MIDASPLUS to tell it when there is an error, but still issues a diagnostic message. In all cases, the VRPG runtime integrity of the MIDASPLUS file is maintained. For example, if a duplicate error occurs after part of the file has been updated, all file data is restored to its prior state before execution halts and further program access to the records is assured.

Compatibility

For compatibility with past revisions of VRPG, the runtime library handles all indexed files that do not have File Specification key continuation statements. These files are handled as they were handled prior to revision 20.0 -- with the primary key only. VRPG programs compiled prior to this release do not have to be recompiled to execute successfully.

ALTERNATE FILE PROCESSING

VRPG provides the following file processing methods to be compatible with IBM System/34 functionality:

- Processing a sequential disk file randomly by relative record number, as if the file is a VRPG direct file
- Processing a MIDASPLUS indexed file randomly by relative record number, as if the file is a VRPG direct file
- Processing a MIDASPLUS indexed file consecutively (read only), as if the file is a VRPG sequential file
- Processing a MIDASPLUS direct file consecutively (read only), as if the file is a VRPG sequential file

Processing a Sequential Disk File Randomly

The file specifications required for a sequential file processed randomly by relative record number are:

Column 15 (file type):	I or U
Column 16 (file designation):	C
Column 19 (file format):	U (if Update)
Column 28 (mode of processing):	R
Column 31 (record address type):	blank
Column 32 (file organization):	D

The file is then read randomly with a CHAIN statement with a relative record number, and optionally updated by exception, detail, or total output. The record number must be a numeric field with length less than, or equal to, 8. If the file is an update file, it must be specified as uncompressed, and the sequential disk file must be uncompressed for the update to occur.

Processing a MIDASPLUS Indexed File Randomly

The file specifications required for a MIDASPLUS indexed file processed randomly by relative record number are:

Column 15 (file type):	I or U
Column 16 (file designation):	C
Column 28 (mode of processing):	R
Column 31 (record address type):	blank
Column 32 (file organization):	D

The file is then read randomly with a CHAIN statement with a relative record number, and optionally updated by exception, detail, or total output. This implementation has a few restrictions:

- Key field in data record. Since the file is really a MIDASPLUS indexed file, the index field of the record may not be changed on an Update operation. It is your responsibility to know which field of the record is the index field. VRPG cannot check this with the current implementation since the program's file specification does not indicate any key location.

- File structure. Accessing an indexed file by the fifth relative record number, for example, actually returns the fifth data record that was added to the MIDASPLUS file. That record may not be the logical fifth record in the file if records were added in an unordered sequence. This action also may not return the same record as accessing a true MIDASPLUS direct file with relative record number of 5 would, because a direct file has record number slots allocated in the order 0, 1, 2, 3, 4, 5, ... That is, relative record number 5 actually could be the sixth record in the file.

- MIDASPLUS data segment size. This implementation assumes a standard default MIDASPLUS data segment (subfile) size. Do not attempt to modify this size.

Processing MIDASPLUS Indexed and Direct Files Consecutively

For a MIDASPLUS indexed or direct file processed consecutively (read only) as a sequential file, the file specifications required are:

Column 15 (file type):	I
Column 16 (file designation):	P, S or D
Column 28 (mode of processing):	blank
Column 31 (record address type):	blank
Column 32 (file organization):	blank

The indexed file is processed by reading the data records consecutively, and bypassing the index. The direct file is processed consecutively by relative record number.

If a file is to be processed in one program as two different files, the read-write lock on the file must be set to N readers and N writers. MIDASPLUS does this during template creation (CREATK). The PRIMOS command RWLOCK <filename> NONE can also do this.

Caution

Use of the alternate file processing features mentioned above requires Revision 20.0 or later of MIDASPLUS. Access of a MIDASPLUS indexed file located on a remote system is handled by the MIDASPLUS installed on the remote system, not by the MIDASPLUS installed on the local system. If the other system has a revision of MIDASPLUS of 19.4 or earlier, then you cannot access the file by relative record number (even though you may have revision 20.0 of MIDASPLUS on your local system). Do not attempt to use this functionality without the correct revision of MIDASPLUS.

10
The MDUMP Utility

This chapter discusses the MDUMP options, the sequential dump file, status, descriptive, and error messages. MDUMP examples are also included.

The MDUMP utility dumps a MIDASPLUS file into a sequential disk file. You can use MDUMP to rebuild existing MIDASPLUS files. Once you have dumped a MIDASPLUS file, you can:

- Edit the resulting sequential file if the data is in ASCII format
- Use the edited file as input to build a new MIDASPLUS file via KBUILD
- Examine the sequential file to see all of a MIDASPLUS file's data records and key values

MDUMP OPTIONS

MDUMP prompts you for the order, contents, and format of the dump that you want to perform. The prompts also ask you how and where MDUMP should record status and error information during the dump.

The following options are available:

Order of the dump: MDUMP can order the output records according to any of the index keys or according to the sequence of the data subfile records. (Data subfile records are stored in the order that they are added to the file and are not necessarily in order according to key value.)

If you use a secondary key to do a dump, only the data records associated with that index are dumped.

Contents of dump: You can specify that you want the output file to contain data records (that is, data subfile entries) only, or index entries only, or both data records and index entries. For example, to check pairs of primary and secondary key values, you might dump only the primary index and one secondary index.

Format of dump: MDUMP produces output in the following formats: BINARY, COBOL, FTNBIN, RPG, or TEXT. (These formats are explained in Chapter 3, BUILDING A MIDASPLUS FILE.)

Log/error file: Whenever you perform a dump, MDUMP displays information about the layout of the output file. It also displays a status report of the dump and any appropriate error messages. If you provide the name of a log/error file, MDUMP writes this information to the file as well to the screen.

Milestone recording: MDUMP displays a status report consisting of a series of entries made periodically during the dump. Each entry includes the current time, the amount of CPU and disk time used so far in the dump, and the number of records that have been processed. The milestone count determines how many of these entries are generated. If MDUMP finds any errors while performing the dump (for example, a record with an invalid primary key), MDUMP reports the errors along with the milestone statistics.

If you choose a milestone count of 0, an entry is made when the dump begins and when the dump ends. If you choose a milestone count that is greater than 0 (N>0), MDUMP makes an entry for every N records processed.

THE SEQUENTIAL DUMP FILE

Each record of the sequential dump file corresponds to one record in the MIDASPLUS file. A record of MDUMP's output can contain the following fields:

- The data subfile entry of the MIDASPLUS record
- The length of the data subfile entry for MIDASPLUS files with variable-length records
- The direct access record number (if the MIDASPLUS file is a direct access file)
- The primary key
- The secondary key (if the dump is a secondary key dump)

MDUMP files do not contain all of the above fields. For example, since direct access files must have fixed-length records, a record number and a data size could not appear in the same MDUMP file. The fields that do appear in the MDUMP file are in the same order as the above list. This order makes the dump file acceptable as input to KBUILD.

The following points concern the sequential dump file:

- If you dump both the data subfile and an index whose keys are embedded in the data, then the keys appear twice in the MDUMP file.
- Sometimes you must dump the primary key. For example, if you want to feed the sequential file to KBUILD to build a new MIDASPLUS file and if the primary key is not in the data record, you must dump the primary key in order to provide KBUILD with the key.
- You cannot dump secondary data.
- MDUMP recognizes and reports on damage that it finds in the MIDASPLUS file being dumped. It includes error messages in the milestone reports. This can be a means of validating the integrity of an index.
- If you believe that your index subfiles have been damaged, and if the keys are embedded in the data, you can use MDUMP and KBUILD to rebuild the file. Dump the data records in the order of the physical storage without dumping any indexes. This step stops MDUMP from referencing the damaged index subfiles.

THE MDUMP DIALOG

Enter MDUMP to begin the MDUMP dialog. The prompts and appropriate user responses are listed below.

Prompt	Response
ENTER TREENAME OF MIDAS FILE TO DUMP:	Enter the pathname of the MIDASPLUS file to be dumped.
ENTER DUMP METHOD ('DATA' OR AN INDEX #):	Enter DATA to dump records in the order of the data subfile records, or enter an index number (0 to 17) to dump records in ascending order by that index.
DO YOU WANT THE DATA RECORD DUMPED?	YES = the data record is dumped. NO = only the index keys are dumped.
DO YOU WANT THE PRIMARY INDEX KEY DUMPED?	YES = primary key is dumped. The key value is appended to the data record if the data record is also being dumped. NO = primary key is not dumped.
DO YOU WANT THE INDEX <#> KEY DUMPED?	This prompt appears only if you specify a secondary index in response to the second prompt. YES = index is dumped. NO = index is not dumped.
ENTER OUTPUT FILE TREENAME:	Enter a filename for the output file. If a file already exists with the name that you chose, MDUMP supplies an error message and asks for another filename.

ENTER OUTPUT FILE FORMAT:	Enter BINARY, COBOL, RPG, or TEXT. If you are unsure, press the carriage return or enter HELP to get a list of these options. These options are explained in Chapter 3, BUILDING A MIDASPLUS FILE.
ENTER LOG/ERROR FILE NAME:	Enter the name of the file to be opened for recording errors and statistics. If this file already exists, it will be overwritten. Press the carriage return if you do not want to open a log/error file. (The statistics and error messages are displayed on the screen as MDUMP executes.)
ENTER MILESTONE COUNT:	Enter a number to indicate how often you want the milestone report to appear. Enter 0 for the briefest version of the status report.

STATUS AND DESCRIPTIVE MESSAGES

MDUMP produces a series of messages describing the status of the dump and the format of its output file. These messages are displayed at your terminal. If you specify a log/error file, all messages are also written to this file. This section describes the dump and the messages that are normally produced.

When you finish the dialog, MDUMP uses your responses to plan the format of the dump file. As it processes, MDUMP produces one message for each field appearing in the output file and tells you what is in each word of an output record. The following is a list of possible messages. (The first three messages always appear.)

1. FORMAT OF MDUMP DUMP FILE: pathname of dump file

2. DUMP FROM MIDASPLUS FILE: pathname of MIDASPLUS file

3. RECORDS ARE record_length WORDS LONG WRITTEN IN format_name FORMAT

   Record_length = length (in words) of the entire MDUMP output record.

   Format_name = BINARY, COBOL, FTNBIN, RPG, or TEXT.

4A. THE DATA PORTION OCCUPIES WORDS 1 THRU x

4B. THE DATA PORTION IS VARIABLE AND OCCUPIES WORDS 1 THRU x

These messages appear only if you are dumping data records. Message 4A appears if the dump file is a text file, and message 4B appears for all other types of dump files. x is the last word that the data occupies.

5A. THE DATA LENGTH IS SPECIFIED AS A ASCII STRING IN BYTES x THRU y

5B. THE DATA LENGTH IS SPECIFIED AS A BINARY STRING IN WORD z

These messages appear only if you are dumping variable-length records. Message 5A appears if the dump file is a text file. x is the starting byte and y is the ending byte of the data length.

Message 5B appears for non-text dump files. BINARY STRING refers to a single-word INTEGER*2 integer. z is the word that contains the data length.

Data length is the length of the data record in the MIDASPLUS file before any padding occurs in the dump. You can use this information to tell the difference between blanks that are part of the data and blanks that pad variable -length records.

6A. THE DIRECT ACCESS RECORD NUMBER IS SPECIFIED AS A ASCII STRING IN BYTES x THRU y

6B. THE DIRECT ACCESS RECORD NUMBER IS SPECIFIED AS A BINARY STRING IN WORDS z THRU w

These messages appear only if you are dumping direct access records in order of DATA or primary key. Message 6A appears if the dump file is a text file. x is the starting byte and w is the ending byte of the direct access record number.

Message 6B appears for non-text dump files. BINARY STRING refers to a two-word REAL*4 floating point number. z is the starting word and w is the ending word of the direct access record number.

If you dump a direct access file by a secondary key or do not dump the data records, the record numbers do not appear in the output file.

7. THE PRIMARY KEY (INDEX 0) IS A key_type KEY IN BYTES x THRU y

This message appears only if you dump the primary key. Key_type is the data type of the key as you defined it in the CREATK session that created the MIDASPLUS file. x is the starting byte and y is the ending byte of the primary key.

8. THE INDEX w KEY IS A key_type KEY IN BYTES x THRU y

   This message appears only if you dump a secondary key. Key_type is the data type of the key as you defined it in the CREATK session that created the MIDASPLUS file. w refers to the index number. x is the starting byte and y is the ending byte of the secondary key.

The dump begins after these messages are produced. MDUMP then:

- Goes through the MIDASPLUS file in the specified order
- Reads in records
- Constructs and writes output records
- Produces milestone reports as you requested them in the dialog

The following is an example of a milestone report with the milestone count set to 1.

```
OK, mdump
[MDUMP rev 19.4.0]

ENTER TREENAME OF MIDAS FILE TO DUMP: bank
ENTER DUMP METHOD ('DATA' OR AN INDEX #): data
DO YOU WANT THE DATA RECORD DUMPED? yes
DO YOU WANT THE PRIMARY INDEX KEY DUMPED? no
ENTER OUTPUT FILE TREENAME: out
ENTER OUTPUT FILE FORMAT: text
ENTER LOG/ERROR FILE NAME: log
ENTER MILESTONE COUNT: 1


FORMAT OF MDUMP DUMP FILE:  <SYS1>COBOL>OUT
DUMP FROM MIDASPLUS FILE: <SYS1>COBOL>BANK

RECORDS ARE     43 WORDS LONG WRITTEN IN 'TEXT  ' FORMAT
THE DATA PORTION OCCUPIES WORDS 1 THRU    43


BEGINNING DUMP
 COUNT      DATE       TIME      CPU MIN   DISK MIN   TOTAL TM      DIFF
     0  05-21-85  15:37:47        0.000      0.000      0.000     0.000
     1  05-21-85  15:37:47        0.003      0.004      0.006     0.006
     2  05-21-85  15:37:47        0.003      0.004      0.007     0.001
DUMP COMPLETE,        2 RECORDS DUMPED
     2  05-21-85  15:37:47        0.004      0.004      0.008     0.001
OK,
```

Since the MILESTONE COUNT was set to 1, a statistics report was generated for each record. Each line shows how many records have been scanned so far (COUNT); the current date and time; the CPU, disk, and total (CPU + disk) time in minutes elapsed since the beginning of the scan; and the increment of total time elapsed since the last milestone.

ERROR MESSAGES

When MDUMP dumps a file, errors that it finds are reported along with the milestone statistics. The following are MDUMP's error messages and their meanings:

- BAD DATA RECORD POINTER - IGNORED

MDUMP found a bad data record pointer in the MIDASPLUS file. The dump continues.

- BAD INDEX BLOCK OR INDEX BLOCK POINTER

MDUMP found an incorrect index block or index block pointer in the MIDASPLUS file. The dump halts.

- UNABLE TO REACH BOTTOM INDEX LEVEL

MDUMP found an incorrect index block or index block pointer before dumping any records. The dump does not occur.

- INDEX BLOCK SIZE GREATER THAN MAXIMUM DEFAULT

MDUMP found an index block larger than the maximum default size. The dump halts.

SAMPLE MDUMP SESSION

This section presents the dialog and output from a sample MDUMP session.

```
BEGINNING DUMP
 COUNT      DATE       TIME      CPU MIN    DISK MIN    TOTAL TM      DIFF
     0   03-10-85   13:25:34       0.000       0.000       0.000       0.000
     1   03-10-85   13:25:35       0.002       0.001       0.003       0.003
     2   03-10-85   13:25:35       0.003       0.001       0.004       0.001
     3   03-10-85   13:25:35       0.003       0.001       0.004       0.001
     4   03-10-85   13:25:35       0.004       0.001       0.005       0.001
DUMP COMPLETE,        4 RECORDS DUMPED
     4   03-10-85   13:25:35       0.005       0.001       0.006       0.001
OK,
```

The OUT file contains the following data:

```
189264289MURRAY, PAUL          MC2837464123 ORCHARD RD   MANCHESTER     NH03102
282765038HARPER, ANNE          CHK412389112 WASHINGTON STNEWTON         MA02159
```

11
Deleting a MIDASPLUS File

This chapter discusses the KIDDEL utility, which provides you with the fastest method of deleting entire MIDASPLUS files or selected indexes of files.

The KIDDEL dialog asks if you want to delete or zero a file's indexes. DELETE gets rid of an entire index subfile while ZERO only deletes the entries and unused space in an index subfile. A zeroed-out file looks exactly like the file's initial template created with CREATK.

Instead of using the KIDDEL utility, you may use the PRIMOS DELETE command to delete an entire MIDASPLUS file.

THE KIDDEL UTILITY

KIDDEL performs the following functions:

- Deletes an entire MIDASPLUS file, including all index subfiles and data subfiles
- Deletes one or more secondary index subfiles
- Deletes work files (called junk files) left over from an aborted MPACK run
- Removes (also known as initializing or zeroing out) all entries from one or more secondary index subfiles

- Removes all entries in the primary and secondary index subfiles; it also removes all entries in the data subfile

KIDDEL DIALOG

This section lists the KIDDEL prompts and the valid responses.

Prompt	Response
FILE NAME	Enter the name of the MIDASPLUS file to be used.
DELETE INDEXES	Enter one of the following: The subfile number (1-17) of one or more of the secondary indexes - deletes the secondary indexes that you listed. (Use commas between the numbers.) ALL - kills the entire file, deletes the file from its directory, and returns you to PRIMOS. JUNK - deletes work area information left over after an aborted MPACK operation. NONE - allows you to zero one or more index subfiles. No index subfiles are deleted.
ZERO INDEXES	This prompt appears only if you entered NONE to the above prompt. Enter one of the following: Numbers of the secondary index subfiles whose entries will be deleted. (Use commas between the numbers.) ALL - zeroes all index subfiles and the data subfile. If it is a direct access file, the file is reinitialized. NONE - returns you to PRIMOS without action.

Note

If you want to delete or zero the primary index, use the ALL response. Do not enter 0 (primary index) to either the DELETE INDEXES or ZERO INDEXES prompts.

KIDDEL ERROR MESSAGES

- FILE IN USE

The file is not available for KIDDEL use. KIDDEL must have exclusive access to the file. You are returned to PRIMOS.

Note

Error messages that are shared by several MIDASPLUS utilities are listed in Appendix B, ERROR MESSAGES.

KIDDEL Examples

The examples below use the BANK file (created in Chapter 2) which was accessed by a BASIC/VM program. KIDDEL is first used to remove entries from the secondary indexes in the file. KIDDEL is then used to zero all of the index subfiles and 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 Chapter 2, CREATING A MIDASPLUS FILE. See Chapter 14, ADDITIONAL CREATK FUNCTIONS, for additional information about USAGE.

Example 1: The following steps eliminate secondary entries:

```
OK, kiddel
[KIDDEL rev 19.4.0]

FILE NAME? bank

DELETE INDEXES: none

ZERO INDEXES: 1,2
```

The following steps check the contents of the index subfiles:

```
OK, creatk
[CREATK rev 19.4.0]

MINIMUM OPTIONS? yes

FILE NAME? bank
NEW FILE? no

FUNCTION? u

INDEX? 0

ENTRIES INDEXED:          0
ENTRIES INSERTED:          4
ENTRIES DELETED:          0
TOTAL ENTRIES IN FILE:          4

LAST MODIFIED BY MIDASPLUS REV. 19.4

FUNCTION? u

INDEX? 1

ENTRIES INDEXED:          0
ENTRIES INSERTED:          0
ENTRIES DELETED:          0
TOTAL ENTRIES IN FILE:          0

LAST MODIFIED BY MIDASPLUS REV. 19.4

FUNCTION? u

INDEX? 2

ENTRIES INDEXED:          0
ENTRIES INSERTED:          0
ENTRIES DELETED:          0
TOTAL ENTRIES IN FILE:          0

LAST MODIFIED BY MIDASPLUS REV. 19.4

FUNCTION? q
```

Example 2: The following steps eliminate all of the index subfile entries and the data subfile entries:

```
OK, kiddel
[KIDDEL rev 19.4.0]

FILE NAME? bank

DELETE INDEXES: none

ZERO INDEXES: all
OK, creatk
[CREATK rev 19.4.0]

MINIMUM OPTIONS? yes

FILE NAME? bank
NEW FILE? no

FUNCTION? u

INDEX? 0

ENTRIES INDEXED:          0
ENTRIES INSERTED:          0
ENTRIES DELETED:          0
TOTAL ENTRIES IN FILE:          0

LAST MODIFIED BY MIDASPLUS REV. 19.4

FUNCTION? u

INDEX? 1

ENTRIES INDEXED:          0
ENTRIES INSERTED:          0
ENTRIES DELETED:          0
TOTAL ENTRIES IN FILE:          0

LAST MODIFIED BY MIDASPLUS REV. 19.4

FUNCTION? q
```

Example 3: The following steps delete an index subfile:

```
OK, kiddel
[KIDDEL rev 19.4.0]

FILE NAME? bank

DELETE INDEXES: 2
OK, creatk
[CREATK rev 19.4.0]

MINIMUM OPTIONS? yes

FILE NAME? bank
NEW FILE? no

FUNCTION? p

INDEX NO.? 2
INDEX DOES NOT EXIST

INDEX NO.? (CR)

FUNCTION? q
OK,
```

12

Cleaning Up a MIDASPLUS File

This chapter discusses the MPLUSCLUP utility. MPLUSCLUP cleans up files (segment directories and subfiles), releases locks held in memory, and cleans up and re-initializes per-user information. The MPLUSCLUP -ALL command also cleans up system information. Only the supervisor terminal operator using the MIDASPLUS DEBUG mode can execute MPLUSCLUP for others.

MPLUSCLUP releases record locks that are recorded in main memory and reports if any record locks are recorded on disk. (To release locks recorded on disk, run the MPACK utility.)

If an error condition abnormally interrupts an application, a static on-unit automatically cleans up for you. The static on-unit closes all MIDASPLUS files that you have opened, releases any held locks, and releases any internal system resources. (Pressing the BREAK key is an example of when the static on-unit is invoked.)

MPLUSCLUP OPTIONS

If you do not specify any options, MPLUSCLUP cleans up for you exclusively. The available MPLUSCLUP options are -USER and -ALL. Both of these options must be issued only from the supervisor terminal. The MIDASPLUS syntax is:

```
                  { -USER user-number }
MPLUSCLUP         {                   }
                  { -ALL              }
```

Use MPLUSCLUP when you or another user receive a fatal error or are forced-logged out and cleanup has not occurred.

To cleanup for another user, issue the

```
MPLUSCLUP -USER user-number
```

command from the supervisor terminal.

The USER option releases all of a specified user's internal resources and all of a user's record locks held in main memory.

ALL cleans up all of the resources except the user file units and the record locks. File units are not closed for the users. MPLUSCLUP waits for all of the users to finish their current operation and then causes the users to pause while MPLUSCLUP cleans up. MPLUSCLUP also reports any users that are hung.

If the MIDASPLUS system hangs while MIDASPLUS is in use, issue the MPLUSCLUP -ALL command from the supervisor terminal. This command releases all of the MIDASPLUS resources for all of the users on the system. (You cannot specify the -ALL option at a user terminal.)

The MPLUSCLUP -ALL command usually restores the system so that all MIDASPLUS users can continue execution from the points where they were interrupted. If MPLUSCLUP cannot complete the clean up, reshare MIDASPLUS.

REMOTE CLEANUP

If you perform MPLUSCLUP without any options, it performs a remote cleanup of slaves for a single user. If you use the -ALL option or the -USER option on a user other than yourself, MPLUSCLUP releases the MIDASPLUS resources for the users on the local system only. To clean up remote slaves created by users other than yourself, run MPLUSCLUP on each node accessed by the applications.

13

Monitoring a MIDASPLUS File

This chapter discusses the menu-driven utility, SPY, which displays certain information at user-supplied intervals.

MIDASPLUS stores information in memory that is used and updated during runtime. This information includes:

- A table of data record locks taken
- System-wide statistics on performance and use of the system
- System-wide configurable parameters
- User-specific configurable parameters
- Keys of locked records for each user

USER INTERFACE

Type SPY to invoke the utility and then choose the appropriate option on the SPY menu. See Figure 13-1 for a sample SPY menu. There are no command line options with SPY. Depending upon your request, a second menu may appear requiring an additional choice. After displaying the information that you requested, SPY allows you to continue at the top level menu again or to stop.

Each menu gives you an opportunity to exit from SPY. Since SPY is a separate offline utility, you may run it whether or not you are using MIDASPLUS, but MIDASPLUS must be initialized on the system.

```
        ENTER                    IF YOU WANT TO:
-------------------------------------------------------------------

         1                 Display  DATA  RECORD  LOCKS

         2                 Display  SYSTEM  STATISTICS

         3                 Display  SYSTEM  CONFIGURATION

         4                 Display  KEYS OF LOCKED RECORDS

         Q                 STOP



Please enter <NUMBER> of the option you choose, or Q[uit] to STOP.
>
```

SPY Menu
Figure 13-1

RECORD LOCKS DISPLAY

MIDASPLUS makes an entry to an in-memory table to hold data record locks. You can display locks

- By user
- By file name (SPY_FNAMES must be ON in the MIDASPLUS configuration file)
- For the entire system

The table holds 8,000 entries. If more than 8,000 record locks are held, the extra locks are not held in memory but are written out to the data record on disk. Because of the 8,000 record limit, there may be record locks that are not included in the memory table and not shown under SPY. SPY prints a message if any record locks are held on disk. The statistics option also displays the number of disk locks taken since initialization and the number currently held on disk.

To see the names of the files with locks, set the SPY_FNAME option ON in the MPLUS.CONFIG file (the default is OFF). The ON option stores the file name in memory when a file is opened. (Only the first sixteen characters of the file name are saved.)

To find out which locks you or other users are holding, select the Display DATA RECORD LOCKS option. Normally, it is not necessary to lock more than a few data records from any file at one time.

STATISTICS DISPLAY

The system keeps statistics as it is running to help you and/or the System Administrator monitor the efficiency of the system's configuration. Statistics are kept on the following information:

- Product
- Buffer management
- Subfile to file unit translation
- Function calls
- Process waits
- Record locks

Product

SPY displays information about the product that includes product name, revision number, level, and date/time last initialized. The message DATE NOT SET may appear if the system date is not set because of a coldstart.

Note

To set the initialization date, reshare MIDASPLUS.

Buffer Management

The MIDASPLUS buffer pool consists of from 2 to 64 available buffers. These buffers are used exclusively for index pages; records are on disks. You may reserve a maximum of 2 buffers at once. MIDASPLUS attempts to reserve the number of buffers it needs when a MIDASPLUS function is invoked. If MIDASPLUS cannot reserve the buffers, it waits until it gets the necessary ones.

The buffer allocation uses a least-recently-used algorithm. The buffer management statistics provide information needed to fine tune the system buffer pool size. These statistics include the number of:

- Requests to get an index block buffer
- Requests to release an index block buffer
- Waits for buffers to become free
- Getbuffs (that is, the number of times that it was necessary to get a buffer) that accessed the same block as last getbuff
- Getbuffs that found the desired block in buffer pool
- Getbuffs that caused a PRWF$$ disk read
- Requests to create a new index block
- Times more than one user used the same buffer at the same time
- PRWF$$ writes of a buffer block
- Times a buffer block was demoted
- Waits for in-transition buffers (that is, buffer is being read or written)
- Cycles through the buffer pool

The product information statistics also include the percentage of

- Calls requiring wait for free buffer
- Times the same buffer as last was hit
- Times desired block was found
- Calls that caused a disk read
- Times multiple users shared a buffer
- Calls that cycled around the buffer pool

Subfile to Fileunit Translation

Getunit statistics include the

- Number of calls to getunit (that is, the number of times that it was necessary to get a file unit)
- Number of times the requested subfile unit was in getunit cache

- Percentage of cache hits
- Number of times all units were used up
- Percentage of times all units were used up

Function Call

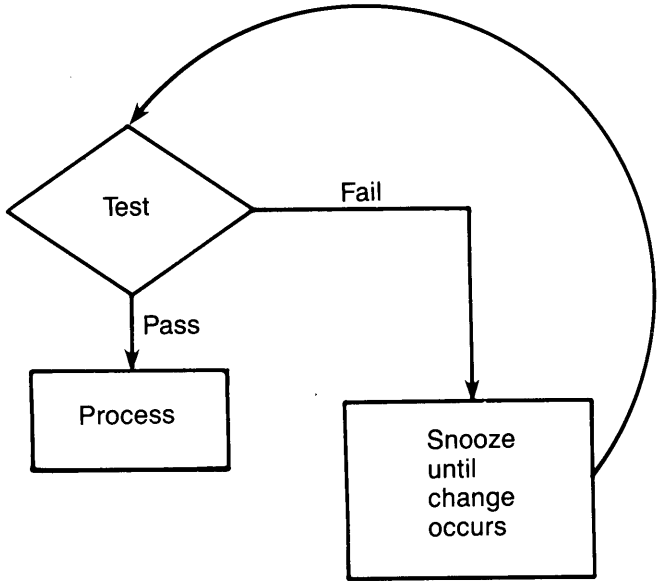
The FUNCTION CALL option tells how many times the following calls occur:

- Local MIDASPLUS calls (OPENM$, CLOSM$, FIND$, FIND$$, NEXT$, NEXT$$, LOCK$, UPDAT$, DELET$, ADDI$, and GDATA$)
- Outgoing remote calls
- Incoming remote calls
- MIDASPLUS remote errors

Process Waits

The PROCESS WAITS option shows the statistics on how often processes must wait to get resources. Since the timeout is configurable, tune the system according to usage. A heavily used system might need a higher timeout value. The terms Snooze and Awaken are used with the PROCESS WAITS INFORMATION option. Snooze causes the user to wait until a particular event occurs; see Figure 13-2. Awaken notifies a user that a condition occurred. The PROCESS WAITS INFORMATION option consists of:

- Number of calls to Snooze and Awaken
- Number of rewaits during Snooze
- Number of timechecks made during a wait
- Number of timeouts due to locked resources
- Average number of timechecks per Snooze
- Average number of rewaits during Snooze



Snooze Flowchart
Figure 13-2

Record Locks

SPY provides the following information about record lock calls:

- Number of record lock calls
- Number of record lock attempts when the record was already locked
- Number of record locks actually written to disk
- Number of records currently locked on disk
- Percentage of calls resulting in disk locks

CONFIGURATION DISPLAY

SPY displays the values of both system-wide and per user configurable parameters. If SYSTEM>MPLUS.CONFIG exists, system parameters are set according to this file; otherwise, default values are assumed.

System Configuration

The system-wide configurable parameters are set when the system is built. The following list defines these parameters:

Parameter	Setting
Debug	ON - prints debug messages
	OFF - no messages (default)
Print error	ON - prints error messages (default)
	OFF - no error messages
Report Dups	ON - reports duplicate key entries (default)
	OFF - no reports
Report Locked	ON - reports the record is already locked on a read operation
	OFF - no reports (default)
Remote Transmit	ON - allows remote calls out (default)
Remote Receive	ON - allows remote calls in (default)
Buffers	Number of buffers from 2 to 64 (default is 64)
Semaphore	Semaphore number that MIDASPLUS uses (default is -14)
Timeout	System wait time (default is 300 seconds)
Funits	Maximum file units that MIDASPLUS can use (default is 256)
Spy_fnames	ON - saves file names when a file is opened
	OFF - (default)
Init_user Common	ON - reinitializes user common information at each new program (command level change). (default)
	OFF - user common information is not reinitialized

Per-user Configuration

The system default acts as the default for the following per-user configurable parameters. They can be configured for an individual user

through MSGCTL. (MSGCTL is described in Chapter 16, INSTALLING AND ADMINISTERING MIDASPLUS.) The following are the per-user configurations when turned on:

Debug	ON - prints debug messages
Print Error	ON - prints error messages
Report Dups	ON - returns duplicate key status
Report Locked	ON - returns record already locked status on reads

Statistics Option Example

The following SPY example displays a summary of the available system statistics and a description of the statistics options.

```
OK, spy
[SPY 22.0]

        ENTER                    IF YOU WANT TO:
 ---------------------------------------------------------------

          1                 Display  DATA  RECORD  LOCKS

          2                 Display  SYSTEM  STATISTICS

          3                 Display  SYSTEM  CONFIGURATION

          4                 Display  KEYS OF LOCKED RECORDS

          Q                 STOP


Please enter <NUMBER> of the option you choose, or Q[uit] to STOP.
> 2
```

SYSTEM STATISTICS MENU

ENTER	IF YOU WANT TO:
1	Display PRODUCT INFO
2	Display BUFFER INFO
3	Display FILE UNITS INFO
4	Display FUNCTION CALLS INFO
5	Display PROCESS WAITS INFO
6	Display RECORD LOCKS INFO
7	Display ALL OF THE ABOVE
8	For HELP - brief description of each option
Q	To STOP

Please enter <NUMBER> of the option you choose, or Q[uit] to STOP.
> 7

PLEASE ENTER LENGTH OF DISPLAY INTERVAL (in 10ths of a second).
IF < 1, STATISTICS WILL BE DISPLAYED ONCE: 1

PLEASE ENTER <NUMBER> OF INTERVALS.
IF < 1, STATISTICS WILL BE DISPLAYED CONTINUOUSLY: 1

STATISTICS WILL BE DISPLAYED EVERY 0.1 SECOND(S) FOR 1 INTERVALS.

Hit RETURN to Continue.

PRODUCT INFO:	PRODUCT NAME:	MIDASPLUS
	REV NUMBER:	19.4.0
	LEVEL:	130
	DATE LAST INITIALIZED:	date not set
FILE UNITS INFO:	TOTAL NUMBER OF CALLS TO GETUNIT:	230
	NUMBER OF GETUNIT CACHE HITS:	49
	PERCENTAGE OF CACHE HITS:	21
	NUMBER OF TIMES UNIT REASSIGNED:	0
	PERCENTAGE OF TIMES UNIT REASSIGNED:	0
PROCESS WAITS INFO:	TOTAL NUMBER OF CALLS TO SNOOZE:	0
	TOTAL NUMBER OF CALLS TO AWAKEN:	15

```
TOTAL NUMBER OF RE-WAITS DURING SNOOZE:        0
AVERAGE NUMBER OF RE-WAITS DURING SNOOZE:      0
TOTAL NUMBER OF TIMECHECKS:                    0
AVERAGE NUMBER OF TIMECHECKS PER SNOOZE:       0
NUMBER OF TIMEOUTS DUE TO LOCKED RESOURCES:    0


                BUFFER   INFORMATION
-------------------------------------------------------------------
    NUMBER OF CALLS TO GET INDEX BLOCK:                    236
    NUMBER OF CALLS TO RELEASE INDEX BLOCK:                60
    NUMBER OF WAITS FOR FREE BUFFER:                       0
    PERCENTAGE OF CALLS REQUIRING WAIT FOR FREE BUFFER:    0
    NUMBER OF TIMES SAME BUFFER AS LAST TIME WAS HIT:      144
    PERCENTAGE OF TIMES SAME BUFFER AS LAST TIME WAS HIT:  61
    NUMBER OF TIMES DESIRED BLOCK WAS FOUND:               75
    PERCENTAGE OF TIMES DESIRED BLOCK WAS FOUND:           31
    NUMBER OF CALLS WHICH CAUSED DISK READ:                17
    PERCENTAGE OF CALLS WHICH CAUSED DISK READ:            7
    NUMBER OF REQUESTS TO CREATE A NEW INDEX BLOCK:        0
    NUMBER OF TIMES MULTIPLE USERS SHARED BUFFER:          0
    PERCENTAGE OF TIMES MULTIPLE USERS SHARED BUFFER:      0
    NUMBER OF DISK WRITES:                                 53
    NUMBER OF TIMES BUFFER WAS DEMOTED:                    0
    NUMBER OF WAITS FOR BUFFERS IN TRANSITION:             0
    NUMBER OF CYCLES AROUND THE BUFFER POOL:               0
    PERCENTAGE OF CALLS WHICH CYCLED AROUND BUFFER POOL:   0


  FUNCTION CALLS INFO:
        NUMBER OF INCOMING REMOTE CALLS:              0
        NUMBER OF OUTGOING REMOTE CALLS:              0
        NUMBER OF MIDASPLUS REMOTE ERRORS:            0
        LOCAL MIDASPLUS CALLS: OPENM$                 13
                               CLOSM$                 5
                               FIND$                  6
                               FIND$$                 0
                               NEXT$                  0
                               NEXT$$                 0
                               LOCK$                  0
                               UPDAT$$                0
                               DELET$                 0
                               ADD1$                  54
                               GDATA$                 0
  RECORD  LOCKS  INFO:
        TOTAL RECORD LOCK CALLS:                          0
        NUMBER OF TIMES RECORD WAS ALREADY LOCKED:        0
        TOTAL NUMBER OF RECORD LOCKS WRITTEN TO DISK:     0
        PERCENTAGE OF CALLS RESULTING IN DISK LOCKS:      0
        CURRENT NUMBER OF RECORDS LOCKED ON DISK:         0

Please enter 1 for SPY menu, 2 for STATISTICS menu, or Q[uit] to STOP
> 2
```

SYSTEM STATISTICS MENU

ENTER	IF YOU WANT TO:
1	Display PRODUCT INFO
2	Display BUFFER INFO
3	Display FILE UNITS INFO
4	Display FUNCTION CALLS INFO
5	Display PROCESS WAITS INFO
6	Display RECORD LOCKS INFO
7	Display ALL OF THE ABOVE
8	For HELP - brief description of each option
Q	To STOP

Please enter <NUMBER> of the option you choose, or Q[uit] to STOP.
> 8

DESCRIPTION OF EACH OPTION

PRODUCT INFO:	Product name, rev number, level number, and date/time last initialized.
BUFFER INFO:	Statistics on index buffer allocation - calls to get or release buffer, waits for free buffers, shared buffers, disk reads and writes, etc.
FILE UNITS INFO:	Total number of calls to getunit, number and percent of: cache hits, times units re-assigned.
FUNCTION CALLS INFO:	Statistics on local Midasplus calls (find$,add1$,etc.) as well as remote calls.
PROCESS WAITS INFO:	Number of snoozes, awakens, and timeouts, total and average rewaits, total and average timechecks.
RECORD LOCKS INFO:	Total record lock calls, calls for already locked records, total disk locks written, percent of calls causing disk locks and current number of disk locks.

Please enter 1 for SPY menu, 2 for STATISTICS menu, or Q[uit] to STOP
> q

SPY is finished
OK,

KEYS OF LOCKED RECORDS DISPLAY

When a user has temporarily left the terminal without releasing a record lock, users who need that record wait for it unnecessarily. If the record has an ASCII primary key and exists in a local file, SPY can show which user is tying up the record. By using the Display KEYS OF LOCKED RECORDS option, you can display the primary key value of each locked record and the number of the user holding it.

In the following example, the file BANK has records locked by two users. User 72 has locked the record with the key value 11, and user 69 has locked the record with the key value 44.

```
OK, spy
[SPY 22.0]

          ENTER                    IF YOU WANT TO:
-------------------------------------------------------------------

           1                 Display  DATA  RECORD  LOCKS

           2                 Display  SYSTEM  STATISTICS

           3                 Display  SYSTEM  CONFIGURATION

           4                 Display  KEYS OF LOCKED RECORDS

           Q                 STOP



Please enter <NUMBER> of the option you choose, or Q[uit] to STOP.
> 4


Enter the FILENAME:
> BANK


          DATA RECORD LOCKS ON FILE <SYS1>EAST>BANK

User No   Record Key
  72      11
  69      44


Please enter F for another FILE, <CR> to CONTINUE or , Q[uit] to STOP
>Q
```

ERRORS

Internal system errors and user input errors are the only kind of errors that can occur during the execution of SPY. Internal system errors are fatal. User input errors can usually be trapped, since only specific input choices are allowed.

If you make a detectable error when entering a menu option, a user number or a file name, you are given two more chances to enter a valid choice and then SPY stops.

If you request that SPY report the number of locks on a file and the SPY_FNAMES configuration is off (the default), the following error message appears:

> The SPY_FNAMES configuration is OFF for MIDASPLUS. SPY cannot display locks by FILENAME. See your System Administrator if you wish to have the SPY_FNAMES configuration changed. Press RETURN to continue.

See Chapter 16, INSTALLING AND ADMINISTERING MIDASPLUS, for additional information about SPY_FNAMES and the configurations.

14
Additional CREATK Functions

Besides creating a template, CREATK allows you to examine and modify a template. This chapter discusses these functions and the extended options feature of CREATK.

FUNCTION SUMMARY

To examine an existing file template, invoke CREATK, provide the file's name or pathname when prompted, and enter NO to the NEW FILE prompt. If you want a list and brief description of the CREATK functions, enter HELP after the FUNCTION prompt, as shown below:

```
OK, creatk
[CREATK rev 19.4.0]

MINIMUM OPTIONS? yes

FILE NAME? bank
NEW FILE? no

FUNCTION? help

A[DD]        =  ADD AN INDEX
C[OUNT]      =  COUNT ACTUAL INDEX ENTRIES
D[ATA]       =  CHANGE DATA RECORD SIZE
E[XTEND]     =  CHANGE SEGMENT & SEGMENT DIRECTORY LENGTH
F[ILE]       =  OPEN A NEW FILE
```

```
G[ET]          =  GET AND SET THE ACTUAL MIN/MAX RECORD SIZE OF THE
                  VARIABLE LENGTH RECORD (VLR) FILE
H[ELP]         =  PRINT THIS SUMMARY
I[NITIALIZE]   =  SET THE MIN/MAX RECORD SIZE FOR THE VARIABLE
                  LENGTH RECORD (VLR) FILE
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]      =  MIDASPLUS DEFAULTS FOR THIS FILE
```

Note

The FILE, HELP, QUIT, and C/R functions are discussed in Chapter 2, CREATING A MIDASPLUS FILE. This chapter discusses the remaining functions in terms of their use for either examining or modifying a file template.

EXAMINING A FILE

The functions to examine a file are:

- COUNT
- PRINT
- SIZE
- USAGE
- VERSION

COUNT

COUNT reads through an index to verify and count each entry. COUNT then displays the total number of valid entries found.

If you request a COUNT on a primary index (index = 0), CREATK determines the number of records inserted or deleted since the last MPACK. If the file descriptor values differ, COUNT updates them.

Inserted/deleted statistics cannot be recreated from a secondary index. Therefore, if there is a difference between the total count and the values of the index descriptor, COUNT is not updated. If this condition occurs, a message appears stating that the file descriptor no longer matches the true state of the file. Use MPACK to reestablish the correct file statistics for display through the USAGE option.

MPACK keeps a count of each time that a record is added to or deleted from a file. These counts are updated at runtime either in intervals of 100 or when the last user closes the file.
If the counters cannot be written to disk, as in the case of a system failure, it is possible that the counts could be off by as much as 100. If this happens, you can correct the counts either by using the COUNT option of CREATK or by using the MPACK utility on the file.

Example:

```
FUNCTION? count
INDEX? 1

TOTAL ENTRIES IN THE INDEX:          4

FUNCTION? count
INDEX? 0

ENTRIES INDEXED:          4
ENTRIES INSERTED:          0
ENTRIES DELETED:          0
TOTAL ENTRIES IN FILE:          4

LAST MODIFIED BY MIDASPLUS REV. 19.4
```

PRINT

PRINT displays index and data subfile information. It displays INDEX NO? to prompt you for an index number. Enter a number from 0 to 17 if you want to see information on a particular index subfile or 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 the last MPACK

PRINT displays the following information for each index level of an index subfile:

- Block size (number of words per block)
- Number of control words
- Maximum number of entries per block

- Length of an index entry
- Number of blocks in that level

Example:

```
FUNCTION? print

INDEX NO.? 1
10 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: 1024 WORDS  # CONTROL WORDS: 10
  MAX ENTRIES/BLOCK:   63    # BLOCKS THIS LEVEL:   1.
```

If you specify the DATA option, CREATK displays:

- File type (keyed-index or direct-access)
- Number of index subfiles defined
- Number of entries currently indexed as of the last MPACK
- Entry size (record size)
- For a variable-length record file, minimum and maximum record sizes, if set
- Primary key size

Example:

The following example shows the use of PRINT with the DATA option on a variable-length record file.

```
FUNCTION? print
INDEX NO.? data
DATA SUBFILE:
   FILE TYPE: KI    # INDEXES:  1   # ENTRIES:      4.
   ENTRY SIZE USER-SUPPLIED
   VLR MIN SIZE:    37     VLR MAX SIZE:    41
   PRIMARY KEY SIZE:     9 BYTES (   5 WORDS)
```

SIZE

Given an expected number of records for a MIDASPLUS file, the SIZE option determines the number of segments and disk records that would be required for an index subfile, a data subfile, or an entire file. After receiving the NUMBER OF ENTRIES prompt, supply the number of records.

After receiving the INDEX NO prompt, enter one of the following:

User Input	System Response
A number from 0 through 17	Estimates size of an individual index subfile
DATA	Estimates size for a data subfile
TOTAL	Estimates an entire file, including all index subfiles and data subfiles
(CR)	Ends the SIZE option dialog and returns you to the FUNCTION? prompt

If you specify 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 assigned for the index blocks already in the index or data subfile

Example:

```
FUNCTION? size
NUMBER OF ENTRIES: 10

INDEX NO.? 1
INDEX  1:        3 440 WD. RECS,        3 1024 WD. RECS
     2 SEGMENTS REQUIRED,  10 SEGMENTS ALLOCATED

INDEX NO.? data
DATA    :        2 440 WD. RECS,        1 1024 WD. RECS
     1 SEGMENTS REQUIRED, 327 SEGMENTS ALLOCATED
```

If the MIDASPLUS file contains variable-length records, the following message appears when you use the DATA option:

VARIABLE LENGTH DATA, NO COMPUTATION

If you specify the TOTAL option in response to the INDEX NO.? prompt, CREATK prints the above information for each index subfile. In addition, it prints the data file plus the number of disk blocks needed to accommodate all index subfiles and the data subfile. For example:

```
INDEX NO.? total
INDEX  0:        3 440 WD. RECS,        3 1024 WD. RECS
     2 SEGMENTS REQUIRED,  10 SEGMENTS ALLOCATED
INDEX  1:        3 440 WD. RECS,        3 1024 WD. RECS
     2 SEGMENTS REQUIRED,  10 SEGMENTS ALLOCATED
INDEX  2:        3 440 WD. RECS,        3 1024 WD. RECS
     2 SEGMENTS REQUIRED,  10 SEGMENTS ALLOCATED
DATA    :        2 440 WD. RECS,        1 1024 WD. RECS
     1 SEGMENTS REQUIRED, 327 SEGMENTS ALLOCATED

TOTAL DISK RECORDS:       11 440 WD. RECS,       10 1024 WD. RECS
```

USAGE

USAGE reads the number of entries that were inserted and deleted since the last MPACK. While COUNT reports the exact number of entries, USAGE reports the number of entries since the last MPACK. USAGE operates from the index description that MIDASPLUS routines maintain and displays the following information:

- Data records indexed as of the last MPACK
- Data records added since the last MPACK
- Data records deleted since the last MPACK
- Total number of data records (entries) in the file
- The version of MIDASPLUS that last modified the file

Example:

```
FUNCTION? usage

INDEX? 1

ENTRIES INDEXED:          4
ENTRIES INSERTED:          0
ENTRIES DELETED:          0
TOTAL ENTRIES IN FILE:          4

LAST MODIFIED BY MIDASPLUS REV. 19.4
```

VERSION

VERSION displays the following information:

- The version of MPLUSLB (the MIDASPLUS library) used in building the template
- The DAM file length (default 524288 words)
- The segment directory length (default 512 segments)
- Maximum segments (subfiles) allocated per index (default 10)
- The maximum number of indexes (including the primary index) that can be defined for the file (default 18)

Example:

```
FUNCTION? version
[CREATK rev 19.4.0]

FILE CREATED BY MPLUSLB REV. 19.4

DEFAULT PARAMETERS FOR FILE
DAM FILE LENGTH       524288 WORDS
BASIC SEGMENT DIRECTORY LENGTH  512SEGMENTS
MAXIMUM SEGMENTS PER INDEX  10
MAXIMUM NUMBER OF INDEXES  18
```

MODIFYING A TEMPLATE

The template modifying functions are:

- ADD
- DATA
- EXTEND
- GET
- INITIALIZE
- MODIFY

Use DATA, EXTEND, and MODIFY only when it is necessary to increase index subfile length or to change the data subfile length. You cannot change key length or key type without recreating the file or the index from the beginning.

Note

Except for GET and INITIALIZE, these options do not take effect until you restructure the file with MPACK.

ADD

ADD allows you to build a secondary index from MIDASPLUS data and increase the number of secondary indexes in your file. Use the BANK file created in Chapter 2, CREATING A MIDASPLUS FILE, as an example. Assume that you want to add a third secondary index. Creating a third index allows you to use the street address from the BANK file as a search key. CREATK prompts you through the ADD function much as it did when you created the initial template.

Remember, you can only define 17 secondary indexes per file. CREATK gives you an error message if you try to create more than 17 secondary indexes or if the secondary index already exists.

Since information is already present in the data subfile record (the entire input record was written to the data subfile), you can tell KBUILD to take the secondary index entries from the data subfile record and add them to secondary index subfile 3. The following example shows how to do this using a MIDASPLUS file that already contains data entries.

```
OK, creatk
[CREATK rev 19.4.0]

MINIMUM OPTIONS? yes

FILE NAME? bank
NEW FILE? no

FUNCTION? add

INDEX NO.? 3

DUPLICATE KEYS PERMITTED? yes

KEY TYPE: a
KEY SIZE = : b 16
SECONDARY DATA SIZE IN WORDS = : (CR)

INDEX NO.? (CR)

FUNCTION? q
OK, kbuild
[KBUILD rev 19.4.0]
```

```
SECONDARIES ONLY? yes
USE MIDASPLUS DATA ENTRIES? yes

ENTER MIDASPLUS FILENAME: bank
SECONDARY KEY NUMBER: 3
ENTER STARTING CHARACTER POSITION: 45
SECONDARY KEY NUMBER: (CR)
IS FILE SORTED? no
ENTER LOG/ERROR FILE NAME: (CR)
ENTER MILESTONE COUNT: 1

DEFERRING: 3

PROCESSING FROM:  bank
 COUNT      DATE       TIME      CPU MIN    DISK MIN    TOTAL TM      DIFF
     0  01-28-85  14:48:35        0.000       0.000       0.000       0.000
     1  01-28-85  14:48:35        0.002       0.001       0.002       0.002
     2  01-28-85  14:48:35        0.002       0.001       0.003       0.001
     3  01-28-85  14:48:35        0.003       0.001       0.003       0.001
     4  01-28-85  14:48:35        0.003       0.001       0.004       0.001
FIRST BUILD/DEFER PASS COMPLETE.
     4  01-28-85  14:48:35        0.004       0.001       0.005       0.001

SORTING INDEX  3
 COUNT      DATE       TIME      CPU MIN    DISK MIN    TOTAL TM      DIFF
     0  01-28-85  14:48:35        0.000       0.000       0.000       0.000
SORT COMPLETE
     4  01-28-85  14:48:36        0.008       0.004       0.012       0.012

BUILDING INDEX   3
 COUNT      DATE       TIME      CPU MIN    DISK MIN    TOTAL TM      DIFF
     0  01-28-85  14:48:36        0.000       0.000       0.000       0.000
     1  01-28-85  14:48:37        0.002       0.000       0.002       0.002
     2  01-28-85  14:48:37        0.003       0.000       0.003       0.001
     3  01-28-85  14:48:37        0.003       0.000       0.004       0.001
     4  01-28-85  14:48:37        0.004       0.000       0.004       0.001
INDEX  3 BUILT
     4  01-28-85  14:48:37        0.005       0.000       0.005       0.001

KBUILD COMPLETE.

OK,
```

DATA

DATA changes the data size or record size in a MIDASPLUS file. Its dialog is similar to the data subfile questions asked during template creation. The new data size becomes effective after you pack the file. (Use the DATA option of MPACK.) When MPACK is run on the modified file, existing records in the file are truncated or padded with zeros to make them compatible with the new data size.

After extending the MIDASPLUS file, use one of the methods listed on Table 14-1 to add records to your file. You should make the file twice as large as the size that you actually need. Doing this allows you to meet the special requirements of ADD1$.

Table 14-1
Methods of Adding Records to a MIDASPLUS File

Method	Action	Advantage	Restrictions
KBUILD	Load a sequential file into a MIDASPLUS file.	Accepts unsorted files.	Only one user at a time can use the file; the file must be empty.
PRIBLD	Initially load a large file.	Loads records quickly and packs records tight.	Input file must be sorted; the file must be empty; only one user at a time can access a file.
ADDI$	Allow multiple users to add records to a file.	Splits the index blocks and leaves room for possible future expansion.	Occasionally you must use MPACK to eliminate the excess disk space.

EXTEND

EXTEND allows you to change the length of the segment subfiles and to extend the length of the segment directory. Supply both the segment directory length and the segment subfile (index) length.

If you enter a 0 or press the RETURN key, CREATK uses the default values. The default subfile size is 1,073,471,824 words, which is the maximum number allowed. This number is the limit to which a subfile can grow before a new subfile is created. Fewer subfiles have the advantage of fewer opens and closes, and thus, fewer file units are used at run time. The minimum segment size is 185 (segments), which reflects a file with only one segment allocated for the data subfile. As the file grows, it uses up more segments.

GET

The GET command performs up to three functions on variable-length record files. First, GET always displays the sizes of the largest and smallest records in the file. Second, if the file has no limits on record sizes, GET sets the limits to the largest and smallest record sizes. Third, if size limits are already set, GET may change them. If the smallest record is smaller than the minimum size, the minimum size changes to the smallest record's size. If the largest record is larger than the maximum size, the maximum size changes to the largest record's size.

To discover the current size limits, use the PRINT command with the DATA option and look for a line with the following format:

```
VLR MIN SIZE:  <number>   VLR MAX SIZE:  <number>
```

If this line is absent, size limits are not set.

Example:

This example shows the use of the GET command, preceded and followed by the use of the PRINT command. In using the PRINT command, the user is looking for the line shown above, which indicates size limits are set. The line is absent in the first use of PRINT.

```
FUNCTION? print

INDEX NO.? data
DATA SUBFILE:
   FILE TYPE: KI     # INDEXES:  1    # ENTRIES:       4.
   ENTRY SIZE USER-SUPPLIED
   PRIMARY KEY SIZE:      9 BYTES (    5 WORDS)
```

Since no information on variable-length records appeared, the user issues the GET command to set the minimum and maximum sizes for these records.

```
FUNCTION? get

PLEASE WAIT WHILE FINDING THE ACTUAL MIN/MAX RECORD SIZE...
ACTUAL MIN RECORD SIZE:     37    ACTUAL MAX RECORD SIZE:     41
```

The user reissues the PRINT command to check that size limits were set to the actual largest and smallest record sizes:

```
FUNCTION? p
```

```
INDEX NO.? data
DATA SUBFILE:
   FILE TYPE: KI    # INDEXES:  1   # ENTRIES:      4.
   ENTRY SIZE USER-SUPPLIED
   VLR MIN SIZE:    37     VLR MAX SIZE:    41
   PRIMARY KEY SIZE:     9 BYTES (   5 WORDS)
```

GET is intended for files containing data. If you use GET on an empty file, CREATK displays:

```
FILE IS EMPTY;  PLEASE USE COMMAND  I[NITIALIZE]  TO  SET  THE  VLR
MIN/MAX SIZE.
```

INITIALIZE

The INITIALIZE command sets the minimum and maximum record sizes in a variable-length record file that is empty. This command also lets you expand these size limits before or after you load the file; you can decrease the minimum record size or increase the maximum record size. However, MIDASPLUS automatically expands size limits whenever you add a record that is outside a defined limit. The exceeded limit becomes the size of the new record.

Example 1:

The following example shows the use of INITIALIZE on a empty file.

```
FUNCTION? initialize

CURRENT MIN/MAX VALUES:     0     0
SET VLR SIZE (MIN/MAX)? 25 39
```

Example 2:

To set or change either size limit, you must specify a minimum of at least 1, and the maximum of at most 32,767. Both values must be specified, even if you are only changing one. This example shows the use of INITIALIZE on a file with a minimum size of 25 and and a maximum size of 49. Desiring to increase the maximum to 51, the user begins by simply entering 51; however, CREATK prompts the user for both sizes.

```
FUNCTION? i

CURRENT MIN/MAX VALUES:    25    49
SET VLR SIZE (MIN/MAX)? 51

SPECIFY THE MINIMUM AND MAXIMUM SIZE OF THE VLR.

CURRENT MIN/MAX VALUES:    25    49
```

```
SET VLR SIZE (MIN/MAX)? 25 51
```

Remember, you can only change size limits by decreasing the minimum or by increasing the maximum. Therefore, if the current limits are 10 and 100, and you enter 10 and 90 as the new limits, CREATK displays:

```
MAX HAS NOT BEEN UPDATED
```

If you use the INITIALIZE command on a file that contains data but has no size limits set, CREATK displays:

```
FILE NOT EMPTY; PLEASE USE COMMAND G[ET] TO SET THE ACTUAL
MIN/MAX SIZE.
```

To discover the current size limits, use the PRINT command with the DATA option and look for a line with the following format:

```
VLR MIN SIZE:  <number>   VLR MAX SIZE:  <number>
```

If this line is absent, size limits are not set.

MODIFY

MODIFY allows you to change the following parameters in an existing MIDASPLUS file template:

- Index block length (only if you are using the extended options path of CREATK)
- Secondary data size (see note below)
- Support for duplicate key occurrences

Notes

When secondary data size is modified for a particular index, the existing secondary data entries are truncated or padded with 0s when the file is packed. This action ensures that all of the secondary data entries in that index conform to the new secondary data size.

If the index that you are trying to MODIFY does not exist, an error message is displayed.

THE EXTENDED OPTIONS PATH

Using the extended options path of CREATK, you can change, on a per-file basis, some of the default file parameters that CREATK uses in initializing MIDASPLUS files. You can also specify the size of an index block at each index level in the index subfile. An index block contains key entries that point to records in the data subfile. An index subfile block entry includes

- A key value, user-supplied during data entry (file building)
- A three-word pointer to a data subfile record (in keyed index files)
- A five-word pointer to a data subfile record (in direct access files)
- Secondary data in secondary indexes (optional)

Defining Block Size

When using minimum options, CREATK automatically supplies you with the default space of 1024 words per block. It is strongly recommended that you define the block space as 1024 words per block for extended options also.

Block Size Specifications

You can change the block size at the first, second, and last index levels (see Index Block Levels below) with the extended options version of CREATK. The minimum acceptable block size must be at least large enough to hold 6 or 10 control words and 2 entries at that particular level. The maximum and suggested block size is 1024 words.

The minimum required block size varies with the level. The last level index block always has ten control words, while upper levels have 10 control words. CREATK checks to see if your proposed block size will accommodate the minimum number of entries and control words, and lets you know if the proposed block size is acceptable. For direct access files, last level index blocks also contain entry numbers. For secondary index subfiles that support the secondary data feature, last level index blocks also contain secondary data.

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 you first allocate space for an index subfile, the subfile has only one index level called

the last level. As the file becomes larger and more complex, more index levels are created to help search efficiency. Blocks in these index levels are collectively called upper level index blocks. Multi-level indexing maximizes search and access efficiency.

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.

Prompt	Response
MINIMUM OPTIONS?	Enter NO.
FILE NAME?	Enter the name of the new file to be created or the name of the existing file to be examined or modified.
NEW FILE?	Enter YES to create a new template. Enter NO to get information about an existing file template.
DIRECT ACCESS?	Enter YES to create a direct access file or NO to create a keyed index file.
DATA SUBFILE QUESTIONS PRIMARY KEY TYPE:	Enter one of the key codes from the MIDASPLUS Key Types Table to define the primary key type.
PRIMARY KEY SIZE = :	Specify size as B nn where nn is the number of bytes for an ASCII key or the number of bits for a bit string key. Specify size as W nn where nn is the number of words for an ASCII key or the number of words for a bit string key.
DATA SIZE IN WORDS = :	For fixed-length records, enter the maximum length in words of the data record in the data subfile. Include the key size in this figure for COBOL files.

	For variable-length records, either press RETURN, or enter 0, or enter 0 followed by a value for the minimum record size and a value for maximum record size.
NUMBER OF ENTRIES TO ALLOCATE?	Enter the number of entries to allocate for the data subfile. (Asked only if you answered YES to the DIRECT ACCESS? prompt.)
FIRST LEVEL INDEX BLOCK SIZE = :	It is recommended that you enter 1024 (the default and maximum size). See Defining Block Size above if you wish to enter another number.
SECOND LEVEL INDEX BLOCK SIZE = :	Enter the same response as for the first level index block.
LAST LEVEL INDEX BLOCK SIZE = :	Enter the same response as for the first level index block.
SECONDARY INDEX	
INDEX NO.?	Enter a number from 1 to 17 or press the RETURN key if you do not want secondary indexes.
DUPLICATE KEYS PERMITTED?	Enter YES or NO. YES allows the same secondary key value to appear more than once in the index.
KEY TYPE:	Enter one of the following codes (A, B, D, I, L, or S). See Chapter 2, CREATING A MIDASPLUS FILE.
KEY SIZE = :	Enter the size of the key in words, bytes, or bits. Size must be preceded by W and a space for words or B and a space for bytes or bits. (Asked only if A or B type key is specified above.)

SECONDARY DATA SIZE IN WORDS = :	For use with FORTRAN, enter the number of words of secondary data to be stored with this secondary key. (Optional) For use with other languages, enter 0 or press the RETURN key.
FIRST LEVEL INDEX BLOCK SIZE = :	Enter the desired number of words per block as for primary index. (See above.)
SECOND LEVEL INDEX BLOCK SIZE = :	Enter the same response as for first level index block.
LAST LEVEL INDEX BLOCK SIZE = :	Enter the same response as for the first level index block.

Note

The secondary index prompts repeat, enabling you to enter information about each secondary index. To complete the CREATK process, press the RETURN key at the INDEX NO? prompt.

Example:

```
OK, creatk
[CREATK rev 19.4.0]

MINIMUM OPTIONS? no

FILE NAME? extendbank
NEW FILE? yes
DIRECT ACCESS? yes

DATA SUBFILE QUESTIONS

PRIMARY KEY TYPE: a
PRIMARY KEY SIZE = : b 9
DATA SIZE IN WORDS = : 44
NUMBER OF ENTRIES TO ALLOCATE? 10

FIRST LEVEL INDEX BLOCK SIZE = : 1024
SECOND LEVEL INDEX BLOCK SIZE = : 1024
LAST LEVEL INDEX BLOCK SIZE = : 1024

SECONDARY INDEX

INDEX NO.? 1
```

```
DUPLICATE KEYS PERMITTED? yes

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

FIRST LEVEL INDEX BLOCK SIZE = : 1024
SECOND LEVEL INDEX BLOCK SIZE = : 1024
LAST LEVEL INDEX BLOCK SIZE = : 1024

INDEX NO.? 2

DUPLICATE KEYS PERMITTED? no

KEY TYPE: a
KEY SIZE = : b 10
SECONDARY DATA SIZE IN WORDS = :  (CR)

FIRST LEVEL INDEX BLOCK SIZE = : 1024
SECOND LEVEL INDEX BLOCK SIZE = : 1024
LAST LEVEL INDEX BLOCK SIZE = : 1024

INDEX NO.?  (CR)

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

15
Packing a MIDASPLUS File

This chapter discusses the MPACK utility. MPACK performs the following functions: recovers data record space that is marked for deletion, increases file efficiency, unlocks records, and restructures index subfiles.

When you delete an index subfile entry, MIDASPLUS automatically recovers the space that the entry formerly occupied. When you delete data subfile records, however, the records are marked for deletion, and then physically removed when the MPACK utility is run.

Note

When making changes to the template structure with the ADD, DATA, EXTEND, or MODIFY options of CREATK, you must use MPACK with the file after specifying the changes. The desired changes do not take effect until MPACK is executed on the file.

FUNCTIONS AND OPTIONS OF MPACK

There are two separate modes of operation for the MPACK utility, the UNLOCK mode and the MPACK mode. The UNLOCK mode unlocks locked data records and the MPACK mode restructures the file. The functions of the MPACK utility are:

- Reclaiming the space that "deleted" records occupy

- Packing of indexes to minimize disk space used
- 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
- Unlocking any of the data records left locked on disk after a program abort or failure

Milestone Reports

Using MPACK, you can open an error/log file to keep track of any errors that occur during UNLOCK or MPACK. You can also record an optional milestone status report for a given number of records. (The milestone count is a user-specified number.) The milestone statistics include date and time (in 24 hour format), CPU and disk time used, and the time expired between the current milestone and the previous one.

UNLOCK Option

The UNLOCK option searches the data subfile entries looking for locked records. The index subfiles are not touched during the UNLOCK option path. Use the UNLOCK option to perform the following:

- Unlock all data records locked on disk.
- Print a total count of records that it unlocked.

Up to 8,000 record locks are recorded in main memory, and any locks in excess of 8000 are recorded on disk. Locks recorded in main memory are not kept across multiple executions of a program. Use record locks only within a single execution of a program.

A lock that is recorded on disk remains in effect until the record is updated or until MPACK cleans up the file. Ending a program and system crashes do not unlock disk-recorded locks (that is, those record locks in excess of the 8000 in main memory).

A record that is locked in main memory is released under the following circumstances:

- The file containing the record is closed.
- The system crashes.
- MIDASPLUS is reshared and initialized on your system.
- The MPLUSCLUP command is issued (with no command line options).

MPACK Mode

Selecting MPACK at the options prompt of the MPACK dialog puts you into the restructure or MPACK mode. The MPACK mode lets you restructure one or more index subfiles, all of the index and data subfiles, or the entire file. During an index restructure operation, MPACK searches the index subfile entries for entries that are marked for deletion or for entries that are out of order. If any keys are out of order, MPACK reports them to you, but does not reorder them. Secondary index entries that point to data subfile entries marked for deletion are deleted, freeing their space for new entries.

In a data subfile restructure, the entries are reordered to correspond to the primary index order. Space occupied by deleted records is reclaimed for use. Original data subfiles are copied to new packed subfiles. At the end of the MPACK run, the original subfiles are deleted and the new ones replace them. Always make sure that MPACK has enough disk space to hold both the original file and the new copy.

The restructure options are: Index-number, ALL, and DATA.

Index-number Option: This option restructures the individual index subfiles that you specify. However, if the index is corrupted or incomplete, MPACK does not remedy the problem. In this case, use MDUMP and then KBUILD on the file.

ALL Option: This option reclaims wasted space from all of the index subfiles and unlocks all of the data records; it does not reclaim space from data records marked for deletion.

DATA Option: This option restructures the data subfile and all indexes.

Use the DATA option to restructure the entire file. Besides checking and reordering the index subfile entries, the DATA option reorders the data subfile entries to correspond to the order of key entries in the primary index subfile. DATA also recovers subfile space occupied by deleted data records. The data subfile is sorted by primary key, using the key order in the primary index subfile. The DATA option makes sequential file processing much faster and key searches more efficient.

If you specify the DATA option, MPACK asks if you want to overwrite the existing file or make a copy and work on it. To make sure that you will always have a copy of the original file in case you need it, answer NO to the OK TO OVERWRITE prompt. If you answer NO, MPACK asks you to specify the name of the file to which you want to write the restructured file. The original file is left in its original state and the changes are made to a copy of the file. If you accidentally specify the name of an existing file, MPACK informs you that the filename already exists.

MPACK DIALOG

The MPACK dialog and the appropriate responses are shown below. Step numbers are added for clarity.

Prompts	Responses
1. ENTER MIDASPLUS FILENAME	Enter the pathname of the existing MIDASPLUS file.
2. 'MPACK' or 'UNLOCK'	Enter MPACK or UNLOCK. If MPACK, make sure that you have enough space to work on two copies of the file. MPACK makes a copy of the file in order to ensure recovery in case of error or abnormal termination. If UNLOCK, the dialog continues at prompt 6.
3. ENTER LIST OF INDEXES, ALL OR DATA:	Enter index number(s) of index subfiles to be packed. Separate the numbers with commas or spaces. Asked only if the MPACK response was given to prompt 2. The dialog continues at prompt 6. Enter ALL to restructure all indexes in the file and unlock all data records. The dialog continues at prompt 6. Enter DATA to restructure the data file and all indexes.
4. OK TO OVERWRITE FILE?	Answer YES or NO. (Asked only if the DATA response was given to prompt 3.) NO = Your file is not overwritten and you will have a copy of your original file. This is the recommended response. YES = the file is restructured and replaced. The dialog continues at prompt 6.
5. NEW FILENAME:	Enter the name that you want MPACK to give to the restructured file. (Asked only if you answered NO to prompt 4.) After packing the file,

	you have the original file intact and a restructured version with the newly specified filename. If you enter the name of an existing file, MPACK returns an error message.
6. ERR/LOG FILE?	Specify an optional error/log filename for errors and milestone counts. Press the RETURN key if you do not want an error log file.
7. MILESTONE COUNT?	Enter the appropriate number of records after which a milestone report will be generated.

After prompt 7, the file is unlocked and restructured. You are returned to PRIMOS command level.

ABNORMAL TERMINATION OF MPACK

If MPACK aborts, the original file is left unchanged whether you are working on the old file or a new copy of it. If you are restructuring a copy of the file without overwriting it, the copy will have partially built indexes and data subfiles. Delete this copy. If you are overwriting an old file when an error or abort occurs, use the JUNK option of KIDDEL to delete all of the partially-used scratch space that MPACK uses during restructuring.

MPACK Examples

The MPACK utility is used in the following two examples on the BANK file that was accessed with BASIC/VM. The first example uses the ALL option and the second example uses the DATA option. The USAGE option of CREATK is used to show the impact that the MPACK utility has on the BANK file.

Example 1: The ALL Option:

```
OK, creatk
[CREATK rev 19.4.0]

MINIMUM OPTIONS? yes

FILE NAME? bank
NEW FILE? no

FUNCTION? u

INDEX? 0

ENTRIES INDEXED:          0
ENTRIES INSERTED:          4
ENTRIES DELETED:          1
TOTAL ENTRIES IN FILE:          3

LAST MODIFIED BY MIDASPLUS REV. 19.4

FUNCTION? u

INDEX? 1

ENTRIES INDEXED:          0
ENTRIES INSERTED:          4
ENTRIES DELETED:          1
TOTAL ENTRIES IN FILE:          3

LAST MODIFIED BY MIDASPLUS REV. 19.4

FUNCTION? u

INDEX? 2

ENTRIES INDEXED:          0
ENTRIES INSERTED:          4
ENTRIES DELETED:          1
TOTAL ENTRIES IN FILE:          3

LAST MODIFIED BY MIDASPLUS REV. 19.4

FUNCTION? q
OK, mpack
[MPACK rev 19.4.0]

FILE NAME? bank

'MPACK' OR 'UNLOCK': mpack
ENTER LIST OF INDEXES, 'ALL', OR 'DATA': all
ENTER LOG/ERROR FILE NAME: (CR)
```

```
ENTER MILESTONE COUNT: 1

BEGINNING PRIMARY INDEX (INDEX 0)
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM     DIFF
     0  05-06-85  09:49:35       0.000      0.000      0.000    0.000
     1  05-06-85  09:49:36       0.002      0.004      0.006    0.006
     2  05-06-85  09:49:36       0.003      0.004      0.007    0.001
     3  05-06-85  09:49:36       0.004      0.004      0.008    0.001
INDEX 0 MPACK COMPLETE,        3. ENTRIES INDEXED
     3  05-06-85  09:49:36       0.005      0.004      0.009    0.002

BEGINNING SECONDARY INDEX  1
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM     DIFF
     0  05-06-85  09:49:36       0.000      0.000      0.000    0.000
     1  05-06-85  09:49:37       0.002      0.001      0.003    0.003
     2  05-06-85  09:49:37       0.002      0.001      0.003    0.001
     3  05-06-85  09:49:37       0.003      0.001      0.004    0.001
INDEX  1 MPACK COMPLETE        3. ENTRIES INDEXED
     3  05-06-85  09:49:37       0.004      0.001      0.006    0.002

BEGINNING SECONDARY INDEX  2
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM     DIFF
     0  05-06-85  09:49:37       0.000      0.000      0.000    0.000
     1  05-06-85  09:49:37       0.002      0.001      0.003    0.003
     2  05-06-85  09:49:37       0.002      0.001      0.004    0.001
     3  05-06-85  09:49:37       0.003      0.001      0.004    0.001
     4  05-06-85  09:49:37       0.004      0.001      0.005    0.001
INDEX  2 MPACK COMPLETE        3. ENTRIES INDEXED
     4  05-06-85  09:49:38       0.005      0.001      0.007    0.002

MPACK COMPLETE.

OK, creatk
[CREATK rev 19.4.0]

MINIMUM OPTIONS? yes

FILE NAME? bank
NEW FILE? no

FUNCTION? u

INDEX? 0
ENTRIES INDEXED:          3
ENTRIES INSERTED:         0
ENTRIES DELETED:          0
TOTAL ENTRIES IN FILE:          3

LAST MODIFIED BY MIDASPLUS REV. 19.4

FUNCTION? u

INDEX? 1
```

```
ENTRIES INDEXED:          3
ENTRIES INSERTED:          0
ENTRIES DELETED:          0
TOTAL ENTRIES IN FILE:          3

LAST MODIFIED BY MIDASPLUS REV. 19.4

FUNCTION? u

INDEX? 2

ENTRIES INDEXED:          3
ENTRIES INSERTED:          0
ENTRIES DELETED:          0
TOTAL ENTRIES IN FILE:          3

LAST MODIFIED BY MIDASPLUS REV. 19.4

FUNCTION? q
```

Example 2: The DATA Option

```
OK, creatk
[CREATK rev 19.4.0]

MINIMUM OPTIONS? yes

FILE NAME? bank
NEW FILE? no

FUNCTION? u

INDEX? 0

ENTRIES INDEXED:          3
ENTRIES INSERTED:          4
ENTRIES DELETED:          0
TOTAL ENTRIES IN FILE:          7

LAST MODIFIED BY MIDASPLUS REV. 19.4

FUNCTION? u

INDEX? 1

ENTRIES INDEXED:          3
ENTRIES INSERTED:          4
ENTRIES DELETED:          0
TOTAL ENTRIES IN FILE:          7
```

```
LAST MODIFIED BY MIDASPLUS REV. 19.4

FUNCTION? u

INDEX? 2

ENTRIES INDEXED:          3
ENTRIES INSERTED:          4
ENTRIES DELETED:          0
TOTAL ENTRIES IN FILE:          7

LAST MODIFIED BY MIDASPLUS REV. 19.4

FUNCTION? q

OK, mpack
[MPACK rev 19.4.0]

FILE NAME? bank

'MPACK' OR 'UNLOCK': mpack
ENTER LIST OF INDEXES, 'ALL', OR 'DATA': data
OK TO OVERWRITE THE FILE? no
ENTER NEW MIDASPLUS FILE NAME: newbank
ENTER LOG/ERROR FILE NAME: xout
ENTER MILESTONE COUNT: 0

BEGINNING PRIMARY INDEX (INDEX 0)
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM     DIFF
     0  05-06-85  10:24:42       0.000      0.000      0.000      0.000
INDEX 0 MPACK COMPLETE,       7. ENTRIES INDEXED
     7  05-06-85  10:24:42       0.004      0.002      0.006      0.006

BEGINNING SECONDARY INDEX  1
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM     DIFF
     0  05-06-85  10:24:42       0.000      0.000      0.000      0.000
INDEX  1 MPACK COMPLETE       7. ENTRIES INDEXED
     7  05-06-85  10:24:42       0.003      0.001      0.004      0.004

BEGINNING SECONDARY INDEX  2
 COUNT      DATE      TIME     CPU MIN   DISK MIN   TOTAL TM     DIFF
     0  05-06-85  10:24:42       0.000      0.000      0.000      0.000
INDEX  2 MPACK COMPLETE       7. ENTRIES INDEXED
     7  05-06-85  10:24:42       0.003      0.000      0.003      0.003

MPACK COMPLETE.
OK, creatk
[CREATK rev 19.4.0]

MINIMUM OPTIONS? yes

FILE NAME? xnewbank
NEW FILE? no
```

```
FUNCTION? u

INDEX? 0

ENTRIES INDEXED:          7
ENTRIES INSERTED:          0
ENTRIES DELETED:          0
TOTAL ENTRIES IN FILE:          7

LAST MODIFIED BY MIDASPLUS REV. 19.4

FUNCTION? u

INDEX? 1

ENTRIES INDEXED:          7
ENTRIES INSERTED:          0
ENTRIES DELETED:          0
TOTAL ENTRIES IN FILE:          7

LAST MODIFIED BY MIDASPLUS REV. 19.4

FUNCTION? u

INDEX? 2

ENTRIES INDEXED:          7
ENTRIES INSERTED:          0
ENTRIES DELETED:          0
TOTAL ENTRIES IN FILE:          7

LAST MODIFIED BY MIDASPLUS REV. 19.4

FUNCTION? q
OK,
```

MPACK ERROR MESSAGES

The following are MPACK error messages. If an error is fatal, MPACK aborts after reporting it. A non-fatal error is a warning only and does not harm the MPACK process.

Fatal Messages

- UNABLE TO REACH BOTTOM INDEX LEVEL

MPACK was unable to find a last level index block for an index. The file is damaged. Use MDUMP to dump the data file into a sequential disk file and use KBUILD to rebuild the file.

- DATA SUBFILE FULL

This message may occur if MPACK is used to implement segment subfiles or segment directories that are smaller than the default. Use the EXTEND option of CREATK to enlarge the subfile size or segment directory length.

- INDEX FULL

This message may occur if MPACK is used to implement index blocks, index subfiles, or segment directories that are smaller than the default. There is no more room in the index subfile. Use the EXTEND option of CREATK to enlarge the subfile size or segment directory length.

- ABORTING MPACK

This message appears when a fatal error occurs. Use the JUNK option of KIDDEL to delete the scratch files created by MPACK, or if you are not overwriting the old file, delete the new file.

- FILE IN USE

This file is not available for MPACK use. MPACK must have exclusive access to the file. You are returned to PRIMOS.

Warning Messages

- INDEX SUBFILE DOES NOT EXIST

You supplied an index number that was not defined for this file.

- FILE ALREADY EXISTS -- TRY AGAIN

You specified the name of an existing file in response to ENTER NEW MIDASPLUS FILE NAME? prompt of the DATA option path. MPACK does not overwrite an existing file in this case. You must enter the name of a non-existent file.

- INVALID KEY SEEN (IGNORED)

A key is out of order in the index or the key is a duplicate and duplicates are not allowed in the specified index.

- INVALID DIRECT ACCESS ENTRY NUMBER SEEN (IGNORED)

A record number is not greater than zero, or is not a whole number, or is greater than the pre-allocated record number limit.

16

Installing and Administering MIDASPLUS

This chapter describes procedures for setting up and monitoring MIDASPLUS. The following topics are discussed:

- Installing and sharing MIDASPLUS
- Initializing MIDASPLUS
- Using MSGCTL
- Networking MIDASPLUS

INSTALLING MIDASPLUS THE FIRST TIME

Depending on whether you have standard MIDASPLUS or execute-only MIDASPLUS, you use one of two command files the first time you install MIDASPLUS. For standard MIDASPLUS, type the following:

```
@@ MIDASPLUS>MIDASPLUS.INITINSTALL.COMI
```

For execute-only MIDASPLUS, type the following:

```
@@ MIDASPLUSEX>MIDASPLUSEX.INITINSTALL.COMI
```

This command file creates the MIDASPLUS* directory and executes the command file that installs the rest of MIDASPLUS. MIDASPLUS needs the MIDASPLUS* directory for error logging. For more details, see SYSTEM ERROR LOGGING later in this chapter.

UPGRADING MIDASPLUS

Depending on whether you have standard MIDASPLUS or execute-only MIDASPLUS, you use one of two command files to upgrade MIDASPLUS to the current revision. For standard MIDASPLUS, type the following:

```
CO MIDASPLUS>MIDASPLUS.INSTALL.COMI
```

For execute-only MIDASPLUS, type the following:

```
CO MIDASPLUSEX>MIDASPLUSEX.INSTALL.COMI
```

Both of the above commands copy MIDASPLUS libraries, programs, and files to appropriate system directories.

SHARING MIDASPLUS

To execute the command file that shares and initializes standard or execute-only MIDASPLUS, type the following:

```
CO SYSTEM>MIDASPLUS.SHARE.COMI
```

This command file shares appropriate segments and installs the sharable MIDASPLUS library. You must run MIDASPLUS.SHARE.COMI each time the system cold starts.

To automatically run this file at each system cold start, edit the system initialization file (either CMDNCO>PRIMOS.COMI or CMDNCO>C_PRMO) by adding the above command.

Once a Rev. 22.0 or higher version of MIDASPLUS is installed, you can have as many MIDASPLUS processes running as PRIMOS allows. Previously, the MIDASPLUS process limit was less than the PRIMOS limit. If you attempt to use more processes than PRIMOS allows, MIDASPLUS still uses the PRIMOS limit.

For further information on installing and sharing MIDASPLUS, see the MIDASPLUS info file.

MIDASPLUS COMPONENTS

MIDASPLUS comes with everything built and ready for installation. MIDASPLUS includes a library of online and offline routines as well as the following utilities:

- CREATK
- KBUILD
- KIDDEL
- MDUMP
- MPACK
- MPLUSCLUP
- SPY

Note

The MIDASPLUS utilities need the private segments 4040 through 4046. Do not use these segments for any other purpose.

An R-mode library (KIDALB) and a synonym library (VKDALB) provide compatibility with previous MIDAS/MIDASPLUS releases.

If you want to use a MIDASPLUS configuration directive, create a file called MPLUS.CONFIG in the directory SYSTEM, adding the configuration directive to this file. Optional configuration directives are discussed later in this chapter.

PROVIDING ACCESS TO MIDASPLUS

To use a utility on a MIDASPLUS file, the file must reside in an Access Control List (ACL) directory. The next higher level directory must also be an ACL directory. The utilities work in a passworded directory only if there is no owner password on the current directory or on the parent directory. For detailed information about ACLs, see the PRIMOS User's Guide.

Before a utility is allowed to access a MIDASPLUS file, the file's RWLOCK is read and then changed to EXCLUSIVE. Users of MIDASPLUS utilities need the following ACL rights:

Protect, Delete, Add, List, Use, Write (PDALUW) on the current directory

List, Use (LU) on the parent directory

In addition to the above ACL rights, all MIDASPLUS users must have at least Access, List, Use, Read, and Write (ALURW) access to MIDASPLUS*. The System Administrator should be given ALL access to MIDASPLUS*. For more details on MIDASPLUS*, see SYSTEM ERROR LOGGING later in this chapter.

INITIALIZING MIDASPLUS

The IMIDASPLUS command, contained in the MIDASPLUS.SHARE.COMI comamnd input file, initializes MIDASPLUS. The IMIDASPLUS command sets configuration parameters, as specified in the MPLUS.CONFIG file, and makes the appropriate initialization call to the MIDASPLUS library. If MPLUS.CONFIG does not exist, IMIDASPLUS uses all system configuration defaults to perform the initialization. For a list of available configuration parameters, type the -HELP option when you execute the SYSTEM>IMIDASPLUS command.

The configuration file contains MIDASPLUS configuration parameters as standard text, stored one parameter per line. If IMIDASPLUS finds an error with a configuration parameter, it prints an error message, assumes the default, and continues initialization. If a parameter is not specified, the default value is assumed.

It is possible to have more than one configuration file. You can tell MIDASPLUS the location of the configuration file you want to use, by specifying the file's pathname when you execute the IMIDASPLUS command. If you do not specify a pathname, IMIDASPLUS uses the MPLUS.CONFIG file in the current directory.

Configuration Parameters

The following paragraphs describe the configuration parameters. A bullet (o) designates the default parameter.

SEMAPHORE semaphore-number

Specifies the PRIMOS semaphore. This number is used for user synchronization.

The default for semaphore-number is -14.

DEBUG { ON / OFF • }

Turns MIDASPLUS debug execution and print options on and off for developer debugging. The default is off.

This parameter sets debug control on a system-wide basis. You can call MSGCTL to set DEBUG on a per-user basis.

FUNITS maximum number of file units

Specifies the maximum number of file units per-user that MIDASPLUS will use for its subfiles. (This number does not include file units that are used for main MIDASPLUS segment directories.) The default is 256 and the maximum is 512.

Do not make the value less than the maximum number of MIDASPLUS segment directories a user is likely to use at once. In most cases, set this value at least four times the desired number. This allows four subfiles per MIDASPLUS file to be open.

TIMEOUT seconds

Specifies the number of seconds that the user will wait for some internal resource (locks or buffers) before MIDASPLUS assumes that the system is hung and aborts the current operation.

Make the seconds argument a positive integer that shows the maximum number of seconds to wait for a resource. If seconds is zero, then no timeout occurs and the user waits indefinitely. The default value is 300 seconds (5 minutes).

PRINT_ERROR { ON • / OFF }

Specifies whether MIDASPLUS should print error messages when fatal errors occur. The default is ON. This parameter sets the error print control on a system-wide basis. A call to MSGCTL allows PRINT_ERROR to be controlled on a per-user basis.

If ON, MIDASPLUS prints the MIDASPLUS error code for any type of fatal error found. If a PRIMOS system call error is found, the system error message is also printed.

If OFF, no error codes are printed.

BUFFERS buffer-count

Specifies the number of internal file buffers that MIDASPLUS will use. The possible values of buffer-count are 2 through 64, with a default of 64.

It is recommended that you do not change this parameter except on systems with few MIDASPLUS users and limited memory.

A low value uses less working-set, but increases the number of system I/Os and user wait time for a buffer. A large value decreases the user wait times, but uses more working-set.

REMOTE_TRANSMIT { ON • / OFF }

Specifies whether MIDASPLUS can access remote files.

If ON (default), allows processing of outgoing requests to other network nodes for access to MIDASPLUS files on their system.

If OFF, does not allow outgoing remote calls.

REMOTE_RECEIVE { ON • / OFF }

Specifies whether remote MIDASPLUS can access local MIDASPLUS files.

If ON (default), allows other network nodes to access MIDASPLUS files on this system.

If OFF, denies remote incoming requests.

REPORT_DUPS { ON • / OFF }

If ON (default), allows MIDASPLUS to report the existence of duplicate key entries. MIDASPLUS returns a value of 1 in the returned status code (word one of the user communication ARRAY) to indicate duplicates. The parameter sets duplicate reporting system-wide. Call MSGCTL to control REPORT_DUPS on a per-user basis.

If OFF, it never returns a status code of 1.

REPORT_LOCKED { ON / OFF • }

Sets the system-wide state for locked record reporting. Call MSGCTL to change REPORT_LOCKED on a per-user basis.

If ON, allows MIDASPLUS to report that a record is locked on a read operation (for example, FIND$, NEXT$).

If OFF (default), sets bit 5 of user communication array word 13 to 0, regardless of the record's locked status.

SPY_FNAMES { ON / OFF • }

Specifies whether to save the filenames of open MIDASPLUS files for use by SPY.

If ON, saves the filenames of open MIDASPLUS files. SPY uses these filenames when displaying which files have record locks.

If OFF (default), filenames are not saved.

INIT_USER_COMMON { ON • / OFF }

If ON, user common information is reinitialized at each new program (command level) change.

If OFF, the user common information is not reinitialized at each new program change. This option is not recommended because it allows MIDASPLUS files to be closed without updating the MIDASPLUS per-user and system-shared information.

SYSLOG { ON / OFF • }

If ON, MIDASPLUS creates daily logs of system error messages. Set SYSLOG to ON when you are experiencing system problems; otherwise, the OFF setting is preferable, since using the error logging mechanism can adversely affect system performance.

If OFF, no new logs are created and no errors are added to any existing log. For more details, see SYSTEM ERROR LOGGING later in this chapter.

SYSTRACE { FMS / USRCOM / UCA / BUFFERS / GDUCA / BCB / DRLCOM / FILCOM / FS / STK / CUR / CLPCOM / ALL }

SYSTRACE is generally intended to aid Prime Customer Service in diagnosing MIDASPLUS system problems. Since adding this parameter may substantially degrade system performance, you should use it only when you are experiencing system problems. Using SYSTRACE has no effect unless SYSLOG is also on. For more details, see SYSTEM ERROR LOGGING later in this in this chapter.

You can use as many directives as you wish, but you must separate them by spaces and place them all on one line.

Each time a system error occurs, MIDASPLUS records in the error log a specific internal MIDASPLUS data structure for each directive you use. The FMS directive records the System Error Variable; the USRCOM directive records miscellaneous USRCOM variables; UCA records the User Communication Array (USRCOM); BUFFERS records per-user buffer information; GDUCA records the GDATA$ user communication array; BCB records the Buffer Control Block; DRLCOM records the Data Record Locking common area; FILCOM records File Common area; FS records the File Table Structure; STK records the MIDASPLUS stack; CUR records the Current Operation Variables; CLPCOM records the Cleanup Common Area. ALL records all of the above.

MSGCTL

MSGCTL is an online MIDASPLUS routine that allows you to control the following on a per-user basis:

- Error message printing
- Debug mode
- Duplicate entry reporting
- Locked record reporting on read operations

The calling sequence of MSGCTL is:

```
CALL MSGCTL(key)
```

where key can have one of the octal values listed below:

Key	Meaning
0	Error printing off
1	Error printing on
2	Debug mode on
4	Debug mode off
10	Duplicate reporting in status word on
20	Duplicate reporting in status word off
40	Locked record reporting in status word on
100	Locked record reporting in status word off

The key values are additive, and you may use multiple key values in the same call. If you supply conflicting values, OFF takes priority over ON.

To specify more than one key value in the same call, add the desired values and use the sum as the key value. For example, a value of 5 (4 + 1) turns the debug mode off and error printing on. Once you set a condition, it remains in effect until you explicitly change it or exit the application.

Examples:

```
CALL MSGCTL(10)     */ sets duplicate reporting on
      .
      .
CALL MSGCTL(2)      */ sets debug mode on, leaves duplicate
                    */ reporting on
CALL MSGCTL(21)     */ sets duplicate reporting off and error
                    */ printing on, leaves debug mode on
```

NETWORKING MIDASPLUS

MIDASPLUS allows each user to access up to 30 remote files on a network assuming the following conditions are met:

- The disk on which the remote file exists is started up on the local system via remote file access (RFA).
- MIDASPLUS is installed on both systems.
- You set MIDASPLUS configuration directives REMOTE_TRANSMIT on the local system and REMOTE_RECEIVE on the remote system to ON. If remote users need to access local files, the local system should also have REMOTE_RECEIVE set to ON.

When you use OPENM$ or NTFYM$ to open a remote file, MIDASPLUS realizes that the file is remote and takes the appropriate action. Remote access is transparent both to you and your application.

Later MIDASPLUS calls accessing that file are routed to the remote system and processed by the NPX slave process on the remote system. The NPX slave process returns the results to the local MIDASPLUS, which returns them to the user. Because this occurs transparently, no special user action is required.

When the MPLUSCLUP command is issued with arguments, no cleanup operations are performed on a remote system. If you need to clean up an operation that performs remote access, run MPLUSCLUP on both the local and the remote systems. Frequently, you will not know which processes need to be cleaned up on the remote system. Enter MPLUSCLUP -ALL to solve this problem.

Note

The maximum record size for a remote file access is 8 K bytes.

SYSTEM ERROR LOGGING

The error logging mechanism is an optional device that automatically creates a system error log each day. A system error log is a file containing all system error messages for one day. An error log file is not provided when MIDASPLUS is installed. The log file is created when a user attempts to write to the file. Activate the error logging mechanism only when you are experiencing system problems, since using this mechanism adversely affects performance.

The error logging mechanism creates a separate log for each day on which at least one system error occurs. If for some reason an error message cannot be added to the current log, MIDASPLUS creates another log to contain that error message only; no other errors are written to this new log. If MIDASPLUS can neither write to the current log nor create an individual error log, an error message is displayed at the user's terminal.

While all kinds of system errors are logged, most of these errors are either:

- Unexpected PRIMOS File Management System errors
- Network errors
- MIDASPLUS system errors
- Corrupted file errors
- Hard-coded file unit errors
- CONDITION errors
- LOCK errors

Following are some numbers and types of common MIDASPLUS errors that may be logged when error logging is activated:

Error Numbers	Error Types
20 - 23	Read/Write Errors
40 - 45	Internal Errors
51, 71, 85, 92	Miscellaneous Errors

For explanations of the above errors, see RUNTIME ERROR CODES in Appendix B.

If a WARM START occurs when error logging is active, MIDASPLUS records a warning message in the error log. Whether or not error logging is active, this message appears on the supervisor terminal.

Error Log Directory and File Names

All system error logs are created in MIDASPLUS*, a new directory created automatically the first time you install either Rev. 20.2 or a higher revision of MIDASPLUS using the new initial installation file. All MIDASPLUS users must have at least Access, List, Use, Read, and Write (ALURW) access to MIDASPLUS*. The System Administrator should be given ALL acccess to MIDASPLUS*.

The daily error logs use the naming format ERROR_LOG_yymmdd, where yymmdd is the date of the log. The individual error logs, created when MIDASPLUS cannot access the current log, use the naming format ERROR_LOG_USERxxx_yymmdd_mmsstt, where xxx is the user number and yymmdd_mmsstt is the date and time of the error.

Enabling Error Logging

If you are currently experiencing system problems, activate the error logging mechanism by performing two consecutive steps:

1. Add the line SYSLOG ON to the MPLUS.CONFIG file.

2. Execute MIDASPLUS.SHARE.COMI or simply type R SYSTEM>IMIDASPLUS.

If you experience problems with error logging, check to see if the MIDASPLUS* directory exists. If MIDASPLUS* does not exist, execute the initial installation command file MIDASPLUS.INITINSTALL.COMI for standard MIDASPLUS or MIDASPLUSEX.INITINSTALL.COMI for execute-only MIDASPLUS. The installation command file creates the MIDASPLUS* directory needed for error logging and installs the rest of MIDASPLUS. When executing the command file, exclude other users from the system.

Error Log Inspection

The section Configuration Parameters earlier in this chapter also describes SYSTRACE, a parameter primarily intended for use by your Prime Customer Service Representative (CSR). The CSR will temporarily add this directive while gathering information relevant to system problems. Adding the SYSTRACE directive can degrade performance substantially. Therefore, when the CSR is not assisting you, you may prefer not to add this directive to MPLUS.CONFIG even if you have SYSLOG ON.

Note

Because SYSTRACE provides information only for errors recorded in the error log, SYSLOG must be set to ON when SYSTRACE is ON. When error logging is no longer necessary, set SYSLOG to OFF. SYSTRACE need not be set to OFF, because it is automatically ignored when SYSLOG is OFF.

17
Offline Create Routines

This chapter discusses KX$CRE and KX$RFC which are user-callable, offline routines that act as an alternative to CREATK. You will find KX$CRE and KX$RFC helpful only if you need to create file templates or read a file configuration from within your application. It is recommended that you use command files that invoke CREATK rather than using KX$CRE or KX$RFC.

KX$CRE

You can use KX$CRE to create a MIDASPLUS file from a program. It is the same routine as that used by CREATK.

KX$CRE Calling Sequence

KX$CRE's calling sequence is:

```
CALL KX$CRE (filnam,namlen,flags,alloc,pridef,secdef,errcod)
```

The arguments used in the above call and their data types are:

Argument	Data Type	Meaning
filnam	INT*2	Pathname of the file to be opened, two characters per word.
namlen	INT*2	Length of filnam in characters.
flags	INT*2	Global flags. See The Flags Argument below.
alloc	REAL*4	Number of data records to pre-allocate if direct access is enabled for this file. Use this argument only if M$DACC is set in flags. Set this argument to 0 when creating a keyed-index file.
pridef(6)	INT*2	Definition array for the primary index. See Table 17-1 and the Pridef and Secdef Arrays section below.
secdef(6,17)	INT*2	Definition array for the 17 secondary indexes. Secdef(1...6,i) contains the definition for secondary index i, where i ranges from 1 to 17. See Table 17-1 and the Pridef and Secdef Arrays section below.
errcod(2)	INT*2	Error code returned by MIDASPLUS. If the error code in errcod(1) is 0, completion was successful. Error code less than 5000 = a file system error. KX$CRE tries to delete the partially created file and ignores any errors incurred in the process. Error code greater than or equal to 5000 = an error in the MIDASPLUS file definition. Errcod(2) contains an index number if applicable. See Non-File System Error Codes below.

The Flags Argument: This argument indicates the file type and the READ/WRITE lock setting of the file to be created in this call.

The following keys set the flags argument:

Key	Function
M$DACC	Enables the file for direct access. Alloc contains the initial number of records to pre-allocate.
M$NRNW	Sets the file lock to n readers and n writers. (You must use this key.)

The Pridef and Secdef Arrays: Assign values to pridef and secdef to indicate the characteristics of the primary index and any secondary indexes that will be included in the file template. Your program passes these arrays to KX$CRE, which uses the information to build the file template.

The six-element one-dimensional array pridef (1...6) contains the necessary information to define the primary index. The two-dimensional secdef array defines the secondary indexes. Secdef(1...6,i) defines secondary index "i". All of the elements in these arrays are INTEGER*2 data type.

Table 17-1 lists the six elements of each array. Table 17-2 divides the flags used for the first element in each of these arrays into three groups. Group one defines special index subfile characteristics; group two defines the key type; and group three tells whether the key size is supplied in bits, bytes, or words.

Select one flag to define the key type and one flag to define the key size. If you are building a secondary index that allows duplicates, you must specify M$DUPP. Assign the flags to the first element 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 has its length defined in words. The actual length is supplied in secdef(2,1).

Table 17-1
Pridef and Secdef Array Elements

Array Elements		
Pridef	Secdef	Description
(1)	(1,i)	Contains one or more flag values specifying the key type and size. For secdef, it also determines the duplicate status of the key. See Table 17-2 for key-type flags.
(2)	(2,i)	States the primary key size in bits, bytes, or words (pridef) or the secondary key size in bits, bytes, or words (secdef). A 0 in secdef indicates that the index does not exist.
(3)		Data record size. Supply a 0 for variable-length records.
	(3,i)	Secondary data size. Supply a 0 if you do not want this feature.
(4)	(4,i)	Level 1 block size. Supply a 0 to use the default block size (1024 words).
(5)	(5,1)	Level 2 block size. Supply a 0 to use the default block size (1024 words).
(6)	(6,i)	Last level block size. Supply a 0 to use the default block size (1024 words).

Table 17-2
Flags for pridef(1) and secdef(1)

Flag Type	Flag Value	Meaning
Index-specific	M$DUPP	Duplicates permitted for this key (secondaries only)
Key	M$BSTR	Bit string
Key	M$SPFP	Single-precision floating point (REAL*4)
Key	M$DPFP	Double-precision floating point (REAL*8)
Key	M$SINT	Short (16 bits) integer (INT*2)
Key	M$LINT	Long (32 bits) integer (INT*4)
Key	M$ASTR	ASCII string
Key size	M$BIT	Key length specified in bits
Key size	M$BYTE	Key length specified in bytes
Key size	M$WORD	Key length specified in words

Non-File System Error Codes

Errors occurring during the building of a template could originate in the file system or in MIDASPLUS. Errors can result from invalid user arguments or an internal MIDASPLUS problem. This section lists the most common KX$CRE error codes.

- ME$BAS

Allocation size is invalid. The number specified in alloc was either less than 1.0, not a whole number, or too big to allocate the number passed in the user supplied alloc argument due to the default segment directory and segment subfile lengths.

- ME$BDS

Data size is invalid because the data size is negative; or the data size specified in pridef(3) indicates variable-length data records, but the file is configured for direct access, and thus requires fixed-length data records.

- ME$BKS

Key size is invalid. For example, the key size is too big, the key size is negative, or the primary key size is 0. (The limit is 16 words except for ASCII strings, which may be up to 32 words.)

- ME$BKT

Key type is invalid.

- ME$BL1

Level 1 block size is invalid. The block size must be positive, not larger than 1024 words, and must hold at least two index entries.

- ME$BL2

Level 2 block size is invalid.

- ME$BLL

Last level block size is invalid. When building a secondary index, this error may also occur 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$NDA

No duplicates are allowed. You specified the flag M$DUPP in pridef(1). Duplicates are never allowed for the primary key.

Note

All of the flags and codes listed above are defined in SYSCOM>PARM.K.INS.FTN

KX$RFC

KX$RFC is a user-callable routine that returns the file configuration of an already existing MIDASPLUS file.

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 follow:

User-Supplied Arguments

Argument	Data Type	Meaning
filnam	INT*2	The pathname of the MIDASPLUS file whose configuration is to be returned.
namlen	INT*2	Length of filnam in characters.

Arguments Returned by KX$RFC

Argument	Data Type	Meaning
flags	INT*2	Global flags: M$DACC is returned for direct access file; 0 for keyed-index file.
alloc	REAL*4	Number of data records pre-allocated if direct access is enabled. Otherwise 0.0 is returned.
pridef(6)	INT*2	Definition array for the primary index.
secdef(6,17)	INT*2	Definition array for the 17 secondary indexes. Secdef(1...6,i) contains the definition for secondary index i.
errcod	INT*2	Error code or 0 if no error.

KX$RFC's arguments are similar to KX$CRE's arguments. With the exception of the differences listed below, refer to the previous KX$CRE discussion for information about using KX$RFC's arguments.

KX$RFC Arguments

You do not supply the flags argument in a call to KX$RFC; a successful call to the KX$RFC subroutine returns it. If the file is a direct access file, the flag M$DACC is returned; otherwise, flags is returned as 0.

Pridef and Secdef Flags: The flags returned on this call are:

Flags	Meaning
M$BSTR, M$MSPFP, M$SINT, M$LINT, and M$ASTR	Possible settings for bits 1-4.
M$DUPP	The settings for bit 6.
MBIT, MBYTE, and M$WORD	Possible values of bits 6-7. (KX$RFC does not usually return M$WORD.)

Element secdef(2,i) is returned as 0 if there is no index i in this file.

Errcod: Unlike KX$CRE, errcod is a one-word argument instead of a two-word array. Error codes less than 5000 indicate file system errors. ME$NMF, which means that the file is not a MIDASPLUS file, is the only code returned greater than 5000. This code could occur for the following reasons:

- The file is not a SAM segment directory.
- Segment subfile 0 (file descriptor subfile) does not exist.
- Segment subfile 0 does not contain the appropriate flags to indicate that the file is a MIDASPLUS file.

18

Offline Build Routines

This chapter discusses the three offline file-building routines which are PRIBLD, SECBLD, and BILD$R. In addition to using KBUILD and ADD1$, a MIDASPLUS file can be built with these offline building routines. You can call these routines from any user program and use them to add index and data subfile entries to keyed-index or direct access MIDASPLUS files. Data entries can have fixed or variable length records and can be sorted or unsorted. Because these routines do not provide the concurrency controls of the other call interface routines, they should be used as program alternatives to KBUILD only to initially populate a file or to add many records at once to an already populated file.

You can call these routines from any Prime-supported language. See the appropriate user manual for information on calling external routines.

Note

When using the routines discussed in this chapter, place the following files into your FORTRAN programs with $INSERT:

```
SYSCOM>PARM.K.INS.FTN
SYSCOM>ERRD.INS.FTN
SYSCOM>KEYS.INS.FTN
```

GUIDELINES

Review the following guidelines before deciding which routines to use for MIDASPLUS file-building.

PRIBLD: Use PRIBLD when all of the following conditions are true:

- You want to build a primary index and data subfile.
- Your data is sorted by primary key.
- The MIDASPLUS file that you are building does not contain any entries. (See note below.)
- There will be no other users in the file when you are accessing it.

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 subfiles to be built do not contain any entries. (See note below.)
- There will be no other users in the file.

Make sure that the input data contains a copy of the primary key associated with each particular secondary key that you want added to the file. SECBLD locates the primary key entry in the index subfile so that the secondary index entry can be added.

Note

If you try to rebuild an existing file that previously contained entries, make sure that each index that you want to build from sorted data is truly empty. Also make sure that the index does not contain obsolete pointers to data subfile entries that no longer exist. Use MPACK or KIDDEL to clean out the index subfiles before trying to rebuild them with sorted input data.

BILD$R: Use BILD$R when one or both of the following conditions occur:

- Your input data is not sorted by the index to be built.
- Your output (MIDASPLUS) file already contains entries in the subfile to be built.

Calls to PRIBLD, SECBLD, and BILD$R can be made from the same program to build one MIDASPLUS file if you do not attempt to build the same index subfile from both BILD$R and PRIBLD or SECBLD at the same time.

RESTRICTIONS

- Only one user at a time may call an offline routine to operate on a file. Once one of these routines opens a file, no one else can invoke any MIDASPLUS routine to work on that file. More than one of these routines can, however, be called from the same program to work on the same file as long as they do not access the same index subfile concurrently. As a result, you can call the file-building routines in the proper sequence to build an entire file from a single program. The second routine to access that subfile will report an error and will abort unless the program provides an alternate return. (See the list of error messages at the end of this chapter.)

- The offline routines check to see that they are not accessing the same index subfile of a MIDASPLUS file simultaneously while building it.

- Programs that use any of these routines to build a MIDASPLUS file cannot be run at the same time as application programs that access the same MIDASPLUS file using the online calls. If any one of the file-building routines opens a MIDASPLUS file, other users or processes cannot use the file for reading or writing.

- PRIBLD, SECBLD, and BILD$R can only be used to build a single MIDASPLUS (output) file. You cannot process more than one output file at a time through any of these file-building routines.

- Never use OPENM$, NTFYM$, or CLOSM$ with these routines.

- When using PRIBLD, SECBLD, or BILD$R, the file should be opened with SRCH$$ or TSRC$$ rather than OPENM$ or NTFYM$.

EVENT SEQUENCE FLAG

PRIBLD, SECBLD, and BILD$R use the same flag argument in their calling sequence. This flag, called seqflg in the argument list, is used as a communications tool between you and the routine. Use seqflg to tell the routine when to start and stop processing. The routine tells you the state of the build operation. The argument can have one of the four values shown in Table 18-1

When first calling one of these routines from a program to add an entry, supply a flag value of 0 in the calling sequence. This is an initialization request to the routine and tells the routine to start

processing the data that your program provided. 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. Your program should then issue a finalization request and set the event sequence flag to 2. (This is done by making another call to the routine in which seqflg has a value of 2. In the finalization request, every other argument in the calling sequence, except for the unit and altrtn and index arguments, is ignored, and may have a value of 0.) Use a finalization request 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 18-1
Event Sequence Flag Values

Value	Explanation
0	You set the value to zero as an initialization request. This signals the routine that the first record of input data is to be processed.
1	The routine sets the value to 1 after the first record has been processed. It remains set at 1 until you set it to 2.
2	You set the value to 2 as a finalization request. The value is passed to the routine after the last entry in input data has been processed. If more than one index subfile is open, make a separate finalization request for each index.
3	The routine sets this value to indicate that finalization is complete. The routine sets the flag to 3 to acknowledge the closing of each index.

Since you can use these routines again for another index, you can reset seqflg to 0 to restart the rebuild process for a new index subfile.

General Use of Sequence Flag: The following is a generalized example of how the event sequence flag is used:

```
C     INITIALIZATION REQUEST
      SEQFLG = 0
      .
      .
      .
C     SET UP ARGUMENTS FOR CALL
      .
      .
      .
      CALL routine-name (SEQFLG, arguments.....)
C     FINISH UP
C     MAKE COMPLETION REQUEST
      SEQFLG =2                          /*REQUEST TO CLOSE SUBFILES
      CALL routine-name (SEQFLG, arguments.....)
```

Error Handling

The use of flags to pass error status codes between the main program and the routines it calls is recommended. Errors are then handled through a normal return to the calling program. The flag is checked and an action is taken, usually by an on-unit (PL/I) or other exception handler.

You can write errors that occur during the file-building process to a disk file instead of having them appear at your terminal. You can call KX$TIM to record milestones in a similar manner. (KX$TIM is discussed in Appendix F, OTHER MIDASPLUS OFFLINE ROUTINES.)

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 in primary key sequence and the MIDASPLUS (output) file must be empty. PRIBLD cannot add sorted primary key entries to an index subfile that already contains values. If the primary index is not empty, use BILD$R.

If PRIBLD adds a variable-length record that is outside a record size limit, MIDASPLUS automatically resets that limit to the size of the record.

PRIBLD Calling Sequence

PRIBLD's calling sequence is:

```
CALL PRIBLD (seqflg, primkey, data, dlength, funit, altrtn, danum)
```

The arguments for PRIBLD are described below.

Argument	Data Type	Definition
seqflg	INT*2	The event sequence flag. See Table 18-1.
primkey	INT*2	A numeric variable or a one-dimensional array, which can be an integer or real number, depending on the key type. It contains the primary key value to use on this call.
data	INT*2	A one-dimensional array containing the data to be added. If dlength is zero, data may also be zero.
dlength	INT*2	The length of data in words. If dlength is less than the record size originally defined for fixed-length record files, the entry written to the MIDASPLUS file will be padded with 0's. Excess data is ignored. For variable-length records, specify the exact length of the record being added.
funit	INT*2	The file unit on which the MIDASPLUS file is opened.
altrtn	INT*2	The number of a statement in the program to be used as an alternate return. If you supply 0 for the altrtn argument, control returns to PRIMOS if an error occurs.
danum	REAL*4	The entry number for direct access files. If the indicated entry slot is already occupied, the entry is added to the end of the data subfile. Specify a 0 for this argument if the MIDASPLUS file being built is a keyed-index file.

SECBLD

The SECBLD routine builds secondary index subfiles from input data that is sorted in secondary key sequence. The index subfile being built must not contain any entries before SECBLD is called. Include a copy of the primary key as one of the arguments in the call. This step allows SECBLD to 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 returns 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,funit,altrtn)
```

There are no special arguments for direct access files. The data entries have already been added and the record numbers do not need to be supplied in order to add secondary index entries. The complete argument list is given below. All arguments are INTEGER*2 data type.

Argument	Definition
seqflg	The event sequence flag, described in Table 18-1.
seckey	The secondary key value to be added to the index subfile.
pkey	The primary key value that references the same record as seckey.
index	The secondary index subfile number being built during this call to SECBLD.
secdat	The secondary data to be stored in this index entry. This applies only to indexes for which the secondary data feature was chosen during index definition. Specify 0 if you do not want to add any secondary data for this index.

sdsiz	The size in words of the secondary data supplied in this call. Specify a 0 if you supplied 0 for the previous argument.
funit	The file unit on which the MIDASPLUS file is open.
altrtn	The alternate return in the calling program to which control is passed if an error occurs. If specified as 0, the program will abort and return to PRIMOS command level.

BILD$R

You can use BILD$R to build the primary index subfile and data subfile, as well as any or all of the secondary index subfiles associated with a particular MIDASPLUS file. BILD$R processes both sorted and unsorted data. It adds entries to files that already contain index entries in primary and/or secondary subfiles and also works on empty MIDASPLUS files.

If BILD$R adds a variable-length record that is outside a record size limit, MIDASPLUS automatically resets that limit to the size of the record.

Do not make calls to BILD$R while calling PRIBLD or SECBLD for the same index. If you make simultaneous calls without an alternate return, the calling program aborts.

BILD$R Calling Sequence

The BILD$R calling sequence is:

```
CALL BILD$R (seqflg,key,pbuf,bufsiz,danum,index,funit,altrtn)
```

The arguments for BILD$R are described below.

Argument	Data Type	Definition
seqflg	INT*2	The event sequence flag. See Table 18-1 above.
key	INT*2	The primary or secondary key value to be added on this call.

pbuf	INT*2	The data subfile entry if you are adding a secondary index entry. The primary key that references the same data record as the secondary key entry being added if you are adding a secondary index entry. If you are using secondary data, place it in pbuf, immediately following the primary key value.
bufsiz	INT*2	The size of pbuf in words. See Note below.
danum	REAL*4	The record entry number for direct access files. Specify a 0 for this argument if the MIDASPLUS file being built is a keyed-index file, or if you are adding a secondary index entry.
index	INT*2	The number of the index subfile being built on this call to BILD$R. (Supply 0 if building the primary index.)
funit	INT*2	The file unit number on which the MIDASPLUS file is opened.
altrtn	INT*2	The alternate return in the calling program to which control is passed if an error occurs. If specified as 0, an error causes the program to abort and returns 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 only the data record. When adding a secondary index entry, bufsiz represents the length of the primary key plus optional secondary data.

OFFLINE ROUTINE EXAMPLE

Suppose that an airline has a file containing flight number, origin, destination, departure time, and arrival time information. Each record has several fields containing this information, but there are no keys by which the file can be searched. Assume that the airline desires to search on a key that is a combination of several fields, for example flight number, origin, and destination and that the airline decides to put this information into a MIDASPLUS file. The flight number, origin,

and destination fields are concatenated to form the primary key, and the date, departure time, and arrival time fields become secondary keys in the MIDASPLUS file. Since KBUILD cannot handle concatenated keys, the offline file-building routines are used to build the MIDASPLUS file. The original sequential disk file is used for input, and it is listed below, along with the CREATK session in which the template was created. Since some of the fields in the input file are sorted and others are not, the sample program uses all three file-building routines.

The Input File: The sequential input file SCHEDULE has 8 fields:

```
195 BOS LOG NWK EWR 02/04/85 12:00 13:00
205 BOS LOG NYC JFK 02/04/85 12:15 12:50
305 BOS LOG NYC LGA 02/04/85 12:30 13:10
696 CHI ORD WOR WOR 02/04/85 10:45 12:15
106 NWK EWR ORL ORL 02/05/85 08:40 11:55
749 NYC LGA CHI ORD 02/05/85 16:00 18:30
650 WOR WOR BOS LOG 02/06/85 12:45 13:15
```

The origin and date fields are the only fields are in sorted order.

The MIDASPLUS Template: The MIDASPLUS file, called FLIGHTS, has four keys which are defined in the following CREATK session:

```
OK, creatk
[CREATK rev 19.4.0]

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 IN WORDS = : 20

SECONDARY INDEX

INDEX NO.? 1

DUPLICATE KEYS PERMITTED? yes

KEY TYPE: a
KEY SIZE = : b 5
SECONDARY DATA SIZE IN WORDS = : (CR)
```

```
INDEX NO.? 2

DUPLICATE KEYS PERMITTED? yes

KEY TYPE: a
KEY SIZE = : b 5
SECONDARY DATA SIZE IN WORDS = : (CR)

INDEX NO.? 3

DUPLICATE KEYS PERMITTED? yes

KEY TYPE: a
KEY SIZE = : b 5
SECONDARY DATA SIZE IN WORDS = : (CR)

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 SCHEDULE is:

```
STATC18.1 OFFLINE ROUTINE PROGRAM.  CHAPTER 18


C     AIRLINE PROGRAM
C
C     THIS PROGRAM BUILDS A MIDASPLUS FILE FROM A SEQUENTIAL DISK FILE
C     USING THE OFFLINE ROUTINES PRIBLD, SECBLD AND BILD$R.
C     THE PRIMARY KEY IS MADE UP OF THREE FIELDS FROM
C     THE INPUT FILE AND IS THUS A CONCATENATED KEY.
C
      INTEGER*2 FILNAM(40),                 /* FILE NAME BUFFER
     +          IFUNIT,                     /* INPUT FILE FUNIT
     +          MFUNIT,                     /* MIDASPLUS OUTPUT FILE FUNIT
     +          PSQFLG,                     /* PRIBLD'S SEQFLG
     +          SSQFLG,                     /* SECBLD'S SEQFLG
     +          BSQFLG(2),                  /* BILD$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.INS.FTN
C
$INSERT SYSCOM>ERRD.INS.FTN
C
$INSERT SYSCOM>PARM.K.INS.FTN
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$$ (K$READ + K$GETU,
     +             FILNAM, IFUNIT, CHRPOS, I, ERRCOD)
      IF (ERRCOD .EQ. E$FNTF) GOTO 5
      IF (ERRCOD .NE. 0) GOTO 9000
      CALL ATTDEV (IFUNIT, 7, IFUNIT, 80)   /* TELL IOCS ABOUT DISK FILE & UNIT
C
C---INPUT AND OPEN THE MIDASPLUS OUTPUT FILE.
C
C   NOTICE THAT (1) IT IS OPENED FOR READING AND WRITING (K$RDWR), AND
C               (2) WE DO NOT (!) CALL NTFYM$ OR OPENM$/CLOSM$
C
20    CALL TNOUA ('ENTER MIDASPLUS OUTPUT FILE NAME: ', 34)
      READ (1, 10) FILNAM
      CHRPOS(1) = 0
      CHRPOS(2) = 80
      MFUNIT = 0
      CALL TSRC$$ (K$RDWR + K$GETU,
     +             FILNAM, MFUNIT, CHRPOS, I, ERRCOD)
      IF (ERRCOD .EQ. E$FNTF) GOTO 20
      IF (ERRCOD .NE. 0) GOTO 9000
C
C---INIT SEQFLG'S
C
      PSQFLG = 0                              /* PRIBLD SEQFLG FOR INDEX 0
      SSQFLG = 0                              /* SECBLD SEQFLG FOR INDEX 1
      BSQFLG(1) = 0                           /* BILD$R SEQFLG FOR INDEX 2
      BSQFLG(2) = 0                           /* 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
100   READ (IFUNIT, 110, END = 500) INPREC  /* READ THE INPUT RECORD
110   FORMAT (20A2)
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, 20, MFUNIT, 0, 0)
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, 0, 0, MFUNIT, 0)
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$R (BSQFLG(1),                  /* NOTE BSQFLG(1) IS FOR INDEX 2.
     +             SECKEY, PRIKEY, 0, 0, 2, MFUNIT, 0)
C
C.....ADD SECONDARY KEY 3 - THE ARRIVAL TIME.
C        THIS FOLLOWS THE SAME PATTERN OF MOVING THE KEY AND
C        CALLING BILD$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(20), 8)
      SECKEY(3) = LS (INPREC(20), 8)
C
      CALL BILD$R (BSQFLG(2),                   /* NOTE BSQFLG(2) IS FOR INDEX 3)
     +             SECKEY, PRIKEY, 0, 0, 3, MFUNIT, 0)
C
      GOTO 100                                  /* LOOP ON READING & ADDING ENTRIES
C
C---INPUT FILE IS EXHAUSTED.
C        SET THE SEQUENCE FLAG TO '2' AND MAKE A FINAL CALL TO THE
C        APPROPRIATE ROUTINE FOR EACH INDEX BEING BUILT; THEN CLOSE
C        THE INPUT AND OUTPUT FILES.
C
500   PSQFLG = 2
      CALL PRIBLD (PSQFLG, 0, 0, 0, MFUNIT, 0, 0) /* FINALIZE PRIMARY KEY
C
      SSQFLG = 2
      CALL SECBLD (SSQFLG, 0, 0, 1, 0, 0, MFUNIT, 0) /* FINALIZE SECONDARY KEY 1
C
      BSQFLG(1) = 2
      CALL BILD$R (BSQFLG(1), 0, 0, 0, 0, 2, MFUNIT, 0) /* FINALIZE SEC. KEY 2
C
      BSQFLG(2) = 2
      CALL BILD$R (BSQFLG(2), 0, 0, 0, 0, 3, MFUNIT, 0) /* FINALIZE SEC. KEY 3
C
      CALL SRCH$$ (K$CLOS, 0, 0, IFUNIT, I, ERRCOD) /* CLOSE INPUT FILE
      IF (ERRCOD .NE. 0) GOTO 9000
C
      CALL SRCH$$ (K$CLOS, 0, 0, MFUNIT, I, ERRCOD) /* CLOSE MIDASPLUS OUTPUT FILE
      IF (ERRCOD .NE. 0) GOTO 9000
C
      CALL EXIT                                 /* EXIT TO PRIMOS
C
C---ERROR HANDLER
C     TAKES THE BRUTE FORCE APPROACH OF CLOSING INPUT & OUTPUT FILES,
C     IGNORING ANY ERRORS ENCOUNTERED, & EXITING WITH A CALL TO ERRPR$.
C
9000  CALL SRCH$$ (K$CLOS, 0, 0, IFUNIT, I, I) /* CLOSE INPUT FILE
C
      CALL SRCH$$ (K$CLOS, 0, 0, MFUNIT, I, I) /* CLOSE MIDASPLUS  OUTPUT FILE
C
      CALL ERRPR$ (K$NRTN, 'EXAMPLE', 6, 0, 0)
C
      END
```

Sample Output: When the program is run, the user enters the name of the input and output MIDASPLUS files. For example:

```
OK, resume airline
ENTER INPUT FILE NAME: schedule
ENTER MIDASPLUS OUTPUT FILE NAME: flights

PRIMARY INDEX AND DATA

SECONDARY INDEX     1
Index 0: Entries indexed:          7
Index  1: Entries indexed:          7
Index  2: Entries indexed:          7
Index  3: Entries indexed:          7
```

PRIBLD, SECBLD, AND BILD$R ERROR MESSAGES

This section lists the PRIBLD, SECBLD, and BILD$R error messages. If you get one of the following error messages, you can call the routine to finish building the index (set SEQFLG to 2). Your file will be built except for the problem that the error code noted. SECBLD and BILD$R replace the symbols ## (used in the error messages below) with a secondary index number.

If a file system error occurs, (an error not listed below) your file may be damaged. If this happens, use KIDDEL to zero your file. Try to figure out what happened and try again.

- Can't Use PRIBLD and BILD$R Simultaneously

You added one or more entries to the primary index with BILD$R and then called PRIBLD. Simultaneous access to the primary index subfile is not allowed. Either continue adding entries or finish building index 0 with the appropriate calls to BILD$R, but not PRIBLD.

- Illegal SEQFLG
- Index ##: Illegal SEQFLG

If either one of the two above messages appears, the value of seqflg is incorrect. The first call to the routine to add an entry must have a seqflg of 0, which the routine returns as a 1. Later calls to the routine to add additional entries must continue to have a seqflg of 1. The final call to the routine to finish building index 0 for that MIDASPLUS file must have a seqflg of 2, which the routine returns as 3.

- Not a Valid MIDASPLUS File

- Index ##: Not a Valid MIDASPLUS File

The first time that the routine is called to add an entry (seqflg = 0) to the primary or secondary index of a MIDASPLUS file, the routine calls KX$RFC to verify that the file is a valid MIDASPLUS file and to gather certain configuration data needed to build the file.

- Index 0: Index Block Size Greater Than Maximum Default

- Index ##: Index Block Size Greater Than Maximum Default

The above two messages indicate that a fatal error may have occurred on the first call to the routine that adds an entry to a file created with the extended options path. The index block size was defined as larger than the default block size of 1024 words. Recreate the file.

- 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 0: ±0.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 preallocated by CREATK. You may have changed this number with CREATK without performing MPACK on the file to effect the change. Use MPACK against the file before restarting.

- Data Subfile Full

- Index 0: Data Subfile Full

If either one of the above two messages appears, no more entries may be added to the data subfile and therefore to the primary index. Call the routine to finish the primary index (with seqflg = 2) with the entries already added.

- Index ##: Does Not Exist

The indicated index is either an invalid index number or does not exist

in the MIDASPLUS file. Either go back, 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

You may have added one or more entries to this index with BILD$R and have now called SECBLD to add an entry to it. You may continue adding entries or finish building this index with the appropriate calls to BILD$R, but not to SECBLD.

- Index ##: Not Zeroed

The index cannot contain any entries if you are trying to use PRIBLD or SECBLD to build it. Use KIDDEL to zero this index or to zero the entire file.

- Index ##: Key Sequence Error

The supplied key is less than the key supplied in the last call to the routine for this index, or the secondary key is a duplicate of the secondary key supplied in the last call to the routine for this index, and the index does not allow duplicates.

- Index ##: Can't Find Primary Key in File

The routine was unable to find the key value supplied for the pbuf or the pkey argument. The primary index subfile does not contain this value.

- Index ##: Index Full

No more entries may be added to this particular secondary index, but you may still call the routine (set seqflg to 2) to finish this index.

- Index ##: Can't Use BILD$R and PRIBLD/SECBLD Simultaneously

This can happen when you have added one or more entries to this index with PRIBLD or SECBLD and then call BILD$R to add an entry to the same index. You can either continue adding entries or make the appropriate calls to either PRIBLD or SECBLD (but not to BILD$R) to finish building this index.

- Index 0: Direct Access File - Index of -1 and Entry # Required for Primary Key

You attempted to build the primary index of a direct access file. Use an index number of -1 (not 0); supply a REAL*4 entry number in danum.

A
Glossary

ADD
: A CREATK option that allows you to add a secondary index subfile and a key to an existing MIDASPLUS template.

awaken
: A condition that notifies you that an event has occurred.

BILD$R
: A MIDASPLUS routine that builds the primary index subfile and data subfile, as well as any or all of the secondary index subfiles associated with a particular MIDASPLUS file.

COBOL status code
: Two-digit number that either indicates that a COBOL operation was successful or describes a condition or problem.

communications array
: An array that stores the following: the current record's address, the current position in the index subfile, a status code for the operation, the word number of the located entry in the index subfile, and the data record address.

COUNT
: A CREATK option that counts the number of entries currently in a file.

CREATK
: A MIDASPLUS utility that creates a template to describe a MIDASPLUS file and allocates space for it.

DATA

1. A CREATK option that changes the data record length and the number of records allocated for a file.

2. The information that is to be written to the data subfile.

3. An MPACK option that restructures an entire file.

data file
Records that can be referenced through an index subfile by specifying a key value.

DELETE
A KIDDEL option that gets rid of an entire index subfile.

dialog
A series of prompts that direct you to supply information to the utility.

direct access
A file access method based on record numbers. Each record in the data subfile is given a unique record number. To access a particular record, you must give MIDASPLUS a record number.

DIRECT files
MIDASPLUS direct access files in VRPG.

duplicate key
More than one secondary key having the same value.

EXTEND
A CREATK option that changes the number of segments per segment directory and words per segment subfile.

extended options dialog
A CREATK dialog with which you supply index block size parameters in place of the minimum options default parameters to build a template.

flag
A switch with a bit value of either on or off that specifies the options for a call.

funit
The file unit on which the MIDASPLUS file is open.

getbuffs
The number of times that it was necessary to get a buffer.

getunit
The number of times that it was necessary to get a file unit.

IMIDASPLUS
A command, contained in the MIDASPLUS.SHARE.COMI cominput file, that initializes MIDASPLUS.

junk files
The work files that are left over from an aborted MPACK run.

KBUILD
A MIDASPLUS utility that adds records to a MIDASPLUS file.

keyed-index access
A file access method based on giving MIDASPLUS a primary or secondary key value and waiting for MIDASPLUS to return the appropriate record.

KIDDEL
A MIDASPLUS utility that deletes an entire MIDASPLUS file or selected indexes of files.

KX$CRE
A user-callable routine that creates a MIDASPLUS file from a program. It is an alternative to CREATK.

KX$RFC
A user-callable routine that returns the file configuration of an already existing MIDASPLUS file.

MDUMP
A MIDASPLUS utility that dumps a MIDASPLUS file into a sequential disk file.

MIDASERR
A BASIC/VM feature that prints BASIC/VM error codes for MIDASPLUS.

MIDASPLUS
The Enhanced Multiple Index Data Access System that is a collection of subroutines and interactive utilities that constructs, accesses, and maintains keyed data files.

MIDASPLUS.INSTALL.COMI
A command input file that places all of the MIDASPLUS files in the appropriate system directories.

MIDASPLUS.SHARE.COMI
A command input file that shares and initializes MIDASPLUS on the system.

minimum options dialog
A CREATK dialog that supplies default parameters for creating a template.

MODIFY
A CREATK option that allows you to change the support of duplicate keys and changes secondary data size.

MPACK
1. A MIDASPLUS utility that recovers data record space that is marked for deletion, increases file efficiency, unlocks records, and restructures index subfiles.

2. The MPACK option that puts you into MPACK Restructure Mode and lets you restructure index subfiles or an entire file.

MPLUSCLUP
A MIDASPLUS utility that cleans up files and subfiles, releases locks held in memory, and cleans up/reinitializes per-user information.

MSGCTL
An online MIDASPLUS routine that allows you to control the following on a per-user basis: error message printing, debug mode, duplicate entry reporting, and locked record reporting on read operations.

partial search
A keyed-index access method that uses the left-most substring of the full key value.

PRIBLD
A MIDASPLUS routine that builds a primary index subfile and adds the corresponding data records to the data subfile.

primary key
A key that must contain a unique value for each record in the data file.

PRINT
A CREATK option that describes each index subfile and the data subfile in terms of segments allocated, index capacity, key type, key size, and the number of index levels for the subfile.

populate
The process of adding data to a MIDASPLUS file.

RELATIVE files
MIDASPLUS direct access files in COBOL.

SECBLD
A MIDASPLUS routine that builds secondary index subfiles from input data that is sorted in primary key sequence.

secondary data
Data that is stored in the secondary index subfile. Since you cannot access secondary data in the same way as ordinary data from the data subfile, the use of secondary is not recommended.

secondary key
A key that is not required to have a unique value for each record in the data file, used for alternative searches. There may be as many as 17 secondary keys per MIDASPLUS file record.

SIZE
A CREATK option that estimates the number of segments and disk records required for a hypothetical number of entries.

snooze
A condition that causes you to wait until a particular event occurs. Used by the SPY utility.

sorted files
Input files that are arranged in ascending order by primary and/or secondary key.

SPY
A MIDASPLUS utility that monitors the following information: data record locks taken, system-wide statistics on performance and use of the system, system-wide configurable parameters, and user-specific configurable parameters.

template
A structural definition of a MIDASPLUS file consisting of all of the index subfiles needed for key value storage.

USAGE
A CREATK option that provides information on the total number of entries in the file, the number of entries indexed, the number of entries deleted, and the number of entries inserted since the last MPACK.

UNLOCK
An MPACK option that unlocks all records locked on disk and prints a total count of unlocked records.

unsorted files
Input files that are not arranged in any particular order.

variable-length record file
A file containing records that vary in size; each record uses only the disk space it needs to contain the data.

VLR file
See variable-length record file

VERSION
A CREATK option that displays the revision stamp of the MIDASPLUS version under which a file was created.

word
16 bits.

ZERO

A KIDDEL option that deletes the entries an unused space in an index subfile. A zeroed out file looks exactly like the file's initial template created with CREATK.

B
Error Messages

This appendix lists the error messages for the MIDASPLUS utilities and offline routines, MIDASPLUS runtime error codes, and the COBOL status codes.

KBUILD ERROR MESSAGES

The following are KBUILD runtime error messages. If an error is fatal, KBUILD aborts after reporting it. Although files are sometimes damaged in fatal errors, the files are usually still usable. A non-fatal error is a warning only and does not harm the KBUILD process. The record that causes the warning message, however, is not added to the file.

- UNABLE TO REACH BOTTOM INDEX LEVEL

The last level index block could not be located; file is damaged. (Fatal)

- FILE IN USE

The file is not available for KBUILD use. KBUILD must have exclusive access to the file. You are returned to PRIMOS. (Fatal)

- INDEX 0 FULL -- INPUT TERMINATED

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

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

If the maximum number of entries in the secondary index is exceeded, KBUILD aborts. Building of other indexes continues. (Fatal, but file is still okay)

- INDEX 0 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. KBUILD is forced to set the delete bit on in data subfile entries whose primary keys do not fit in the primary index. (Fatal, but file is still okay)

- INVALID DIRECT ACCESS ENTRY NUMBER -- RECORD NOT ADDED

The user-supplied direct access record number is an ASCII string, but it is not legitimate if it contains non-numeric characters. Also, the entry number may be less than or equal to 0, may not be a whole number or may exceed the 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. Do not specify the sorted option during the KBUILD dialog. (Non-fatal)

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

This error occurs when adding secondary index entries to an already populated file. The primary key value that you supplied in the input file was not found in the primary index. (Non-fatal)

- INDEX 0: INVALID KEY -- RECORD NOT ADDED

This error 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 does not allow duplicates. (Non-fatal)

- INDEX { 0 / index-no: } KEY SEQUENCE ERROR -- RECORD NOT ADDED

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

MDUMP STATUS AND DESCRIPTIVE MESSAGES

MDUMP produces a series of messages describing the status of the dump and the format of its output file. These messages are displayed at your terminal. If you specify a log/error file, all messages are also written to this file. This section describes the dump and the messages that are normally produced.

When you finish the dialog, MDUMP uses your responses to plan the format of the dump file. As it processes, MDUMP produces one message for each field appearing in the output file and tells you what is in each word of an output record. The following is a list of possible messages. (The first three messages always appear.)

1. FORMAT OF MDUMP DUMP FILE: pathname of dump file

2. DUMP FROM MIDASPLUS FILE: pathname of MIDASPLUS file

3. RECORDS ARE record_length WORDS LONG WRITTEN IN format_name FORMAT

   Record_length = length (in words) of the entire MDUMP output record.

   Format_name = BINARY, COBOL, FTNBIN, RPG, or TEXT.

4A. THE DATA PORTION OCCUPIES WORDS 1 THRU x

4B. THE DATA PORTION IS VARIABLE AND OCCUPIES WORDS 1 THRU x

   These messages appear only if you are dumping data records. Message 4A appears if the dump file is a text file, and message 4B appears for all other types of dump files. x is the last word that the data occupies.

5A. THE DATA LENGTH IS SPECIFIED AS A ASCII STRING IN BYTES x THRU y

5B. THE DATA LENGTH IS SPECIFIED AS A BINARY STRING IN WORD z

These messages appear only if you are dumping variable-length records. Message 5A appears if the dump file is a text file. x is the starting byte and y is the ending byte of the data length.

Message 5B appears for non-text dump files. "BINARY STRING" refers to a single-word INTEGER*2 integer. Z is the word that contains the data length.

Data length is the length of the data record in the MIDASPLUS file before any padding occurs in the dump. You can use this information to tell the difference between blanks that are part of the data and blanks that pad variable -length records.

6A. THE DIRECT ACCESS RECORD NUMBER IS SPECIFIED AS A ASCII STRING IN BYTES x THRU y

6B. THE DIRECT ACCESS RECORD NUMBER IS SPECIFIED AS A BINARY STRING IN WORDS z THRU w

These messages occur only if you are dumping direct access records in order of DATA or primary key. Message 6A appears if the dump file is a text file. x is the starting byte and w is the ending byte of the direct access record number.

Message 6B appears for non-text dump files. BINARY STRING refers to a two-word REAL*4 floating point number. z is the starting word and w is the ending word of the direct access record number.

If you dump a direct access file by a secondary key or do not dump the data records, the record numbers do not appear in the output file.

7. THE PRIMARY KEY (INDEX 0) IS A key_type KEY IN BYTES x THRU y

This message occurs only if you dump the primary key. Key_type is the data type of the key as you defined it in the CREATK session that created the MIDASPLUS file. x is the starting byte and y is the ending byte of the primary key.

8. THE INDEX w KEY IS A key_type KEY IN BYTES x THRU y

This message occurs only if you dump a secondary key. Key_type is the data type of the key as you defined it in the CREATK session that created the MIDASPLUS file. w refers to the index number. x is the starting byte and y is the ending byte of the secondary key.

MDUMP ERROR MESSAGES

When MDUMP dumps a file, errors that it finds are reported along with the milestone statistics. The following are MDUMP's error messages and their meanings:

- BAD DATA RECORD POINTER - IGNORED

MDUMP found a bad data record pointer in the MIDASPLUS file. The dump continues.

- BAD INDEX BLOCK OR INDEX BLOCK POINTER

MDUMP found an incorrect index block or index block pointer in the MIDASPLUS file. The dump halts.

- UNABLE TO REACH BOTTOM INDEX LEVEL

MDUMP found an incorrect index block or index block pointer before dumping any records. The dump does not occur.

- INDEX BLOCK SIZE GREATER THAN MAXIMUM DEFAULT

MDUMP found an index block larger than the maximum default size. The dump halts.

KIDDEL ERROR MESSAGES

- FILE IN USE

The file is not available for KIDDEL use. KIDDEL must have exclusive access to the file. You are returned to PRIMOS.

SPY ERRORS

Internal system errors and user input errors are the only kind of errors that can occur during the execution of SPY. Internal system errors are fatal. User input errors can usually be trapped since only specific input choices are allowed.

If you make a detectable error when entering a menu option, you are given two more chances to enter a valid choice and then SPY stops. If an out of range or otherwise detectable invalid entry is made for user number or filename, you are given two more chances before SPY gives up.

If you request that SPY report the number of locks on a file and the SPY_FNAMES configuration is off (the default), the following error message occurs.

> The SPY_FNAMES configuration is OFF for MIDASPLUS. SPY cannot display locks by FILENAME. See your System Administrator if you wish to have the SPY_FNAMES configuration changed. Hit RETURN to Continue.

See Chapter 16, INSTALLING AND ADMINISTERING MIDASPLUS, for additional information about SPY_FNAMES and the configurations.

MPACK ERROR MESSAGES

The following are MPACK error messages. If an error is fatal, MPACK aborts after reporting it. A non-fatal error is a warning only and does not harm the MPACK process.

Fatal Messages

- UNABLE TO REACH BOTTOM INDEX LEVEL

MPACK was unable to find a last level index block for an index. The file is damaged. Use MDUMP to dump the data file into a sequential disk file and use KBUILD to rebuild the file.

- DATA SUBFILE FULL

This message may occur if MPACK is used to implement segment subfiles or segment directories that are smaller than the default. Use the EXTEND option of CREATK to enlarge the subfile size or segment directory length.

- INDEX FULL

This message may occur if MPACK is used to implement index subfiles or segment directories that are smaller than the default. There is no more room in the index subfile. Use the EXTEND option of CREATK to enlarge the subfile size or segment directory length.

- ABORTING MPACK

This message appears when a fatal error occurs. Use the JUNK option of KIDDEL to delete the scratch files created by MPACK, or if you are not overwriting the old file, delete the new file.

- FILE IN USE

This file is not available for MPACK use. MPACK must have exclusive access to the file. You are returned to PRIMOS.

Warning Messages

- INDEX SUBFILE DOES NOT EXIST

You supplied an index number that was not defined for this file.

- FILE ALREADY EXISTS -- TRY AGAIN

You specified the name of an existing file in response to ENTER NEW MIDASPLUS FILE NAME? prompt of the DATA option path. MPACK does not overwrite an existing file in this case. You must enter the name of a non-existent file.

- INVALID KEY SEEN (IGNORED)

A key is out of order in the index or the key is a duplicate and duplicates are not allowed in the specified index.

- INVALID DIRECT ACCESS ENTRY NUMBER SEEN (IGNORED)

A record number is not greater than zero, or is not a whole number, or is greater than the pre-allocated record number limit.

KXCRE ERROR MESSAGES

Errors occurring during the building of a template could originate in the file system or in MIDASPLUS. Errors can result from invalid user arguments or an internal MIDASPLUS problem. This section lists the most common KX$CRE error codes.

- ME$BAS

Allocation size is invalid. The number specified in alloc was either less than 1.0, not a whole number, or too big to allocate the number passed in the user supplied alloc argument due to the default segment directory and segment subfile lengths.

- ME$BDS

Data size is invalid because the data size is negative; or the data size specified in pridef(3) indicates variable-length data records, but the file is configured for direct access, and thus requires fixed-length data records.

- ME$BKS

Key size is invalid. For example, the key size is too big, the key size is negative, or the primary key size is 0. (The limit is 16 words except for ASCII strings, which may be up to 32 words.)

- ME$BKT

Key type is invalid.

- ME$BL1

Level 1 block size is invalid. The block size must be positive, not larger than 1024 words, and must hold at least two index entries.

- ME$BL2

Level 2 block size is invalid.

- ME$BLL

Last level block size is invalid. When building a secondary index, this error may also occur 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$NDA

No duplicates are allowed. You specified the flag M$DUPP in pridef(1). Duplicates are never allowed for the primary key.

PRIBLD, SECBLD, AND BILD$R ERROR MESSAGES

This section lists the PRIBLD, SECBLD, and BILD$R error messages. If you get one of the following error messages, you can call the routine to finish building the index (set seqflg to 2). Your file will be built except for the problem that the error code noted. SECBLD and BILD$R replace the symbols ## (used in the error messages below) with a secondary index number.

If a file system error occurs, (an error not listed below) your file may be damaged. If this happens, use KIDDEL to zero your file. Try to figure out what happened and try again.

- Can't Use PRIBLD and BILD$R Simultaneously

You added one or more entries to the primary index with BILD$R and then called PRIBLD. Simultaneous access to the primary index subfile is not allowed. Either continue adding entries or finish building index 0 with the appropriate calls to BILD$R, but not PRIBLD.

- Illegal SEQFLG
- Index ##: Illegal SEQFLG

If either one of the two above messages appears, the value of seqflg is incorrect. The first call to the routine to add an entry must have a seqflg of 0, which the routine returns as a 1. Later calls to the routine to add additional entries must continue to have a seqflg of 1. The final call to the routine to finish building index 0 for that MIDASPLUS file must have a seqflg of 2, which the routine returns as a 3.

- Not a Valid MIDASPLUS File
- Index ##: Not a Valid MIDASPLUS File

The first time that the routine is called to add an entry (seqflg = 0) to the primary or secondary index of a MIDASPLUS file, the routine calls KX$RFC to verify that the file is a valid MIDASPLUS file and to gather certain configuration data needed to build the file.

- Index 0: Index Block Size Greater Than Maximum Default
- Index ##: Index Block Size Greater Than Maximum Default

The above two messages indicate that a fatal error may have occurred on the first call to the routine that adds an entry to a file created with the extended options path. The index block size was defined as larger than the default block size of 1024 words. Recreate the file.

- 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 0: ±0.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 preallocated by CREATK. You may have changed this number with CREATK without performing MPACK on the file to effect the change. Use MPACK against the file before restarting.

- Data Subfile Full

- Index 0: Data Subfile Full

If either one of the above two messages appears, no more entries may be added to the data subfile and therefore to the primary index. Call the routine to finish the primary index (with seqflg = 2) with the entries already added.

- Index ##: Does Not Exist

The indicated index is either an invalid index number or does not exist in the MIDASPLUS file. Either go back, 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

You may have added one or more entries to this index with BILD$R and have now called SECBLD to add an entry to it. You may continue adding entries or finish building this index with the appropriate calls to BILD$R, but not to SECBLD.

- Index ##: Not Zeroed

The index cannot contain any entries if you are trying to use PRIBLD or SECBLD to build it. Use KIDDEL to zero this index or to zero the entire file.

- Index ##: Key Sequence Error

The supplied key is less than the key supplied in the last call to the routine for this index, or the secondary key is a duplicate of the secondary key supplied in the last call to the routine for this index, and the index does not allow duplicates.

- Index ##: Can't Find Primary Key in File

The routine was unable to find the key value supplied for the pbuf or pkey argument. The primary index subfile does not contain this value.

- Index ##: Index Full

No more entries may be added to this particular secondary index, but you may still call the routine (set seqflg to 2) to finish this index.

- Index ##: Can't Use BILD$R and PRIBLD/SECBLD Simultaneously

This can happen when you have added one or more entries to this index with PRIBLD or SECBLD and then call BILD$R to add an entry to the same index. You can either continue adding entries or make the appropriate calls to either PRIBLD or SECBLD (but not to BILD$R) to finish building this index.

- Index 0: Direct Access File - Index of -1 and Entry # Required for Primary Key

You attempted to build the primary index of a direct access file. Use an index number of -1 (not 0); supply a REAL*4 entry number in danum.

RUNTIME ERROR CODES

The following is a list and explanation of the MIDASPLUS runtime error codes. The error codes are returned directly to you unless error traps are included in your program. If an error is not listed, check Appendix C, PRIMOS ERROR MESSAGES. If you are using COBOL, check to see if the error is a MIDASPLUS error or a file status error. (COBOL status codes are listed later in this appendix.)

Miscellaneous Error Codes

Code	Explanation
1	Duplicates exist for the current key.
7	The sought-after entry does not exist in the file.
10	Locking was requested on a record that is already locked. COBOL status code is 90. Check your ACLS and make sure that the READ/WRITE locks are set correctly. The record might not be locked; you could lack the necessary ACLs.
11	The data record does not have the locked bit set when it should. This happens when a user attempts an update without first locking the record.
12	Duplicate keys are not allowed.
13	An unrecoverable concurrency error has occurred. For example, another user deleted your current entry.
19	The disk is full.

READ/WRITE Error Codes

Error codes in the 20 range are usually READ/WRITE errors. Try to do a LOGPRT which creates a LOGLST. This will tell you if there are any unrecoverable errors on disk or memory parity errors. (See the System Operator's Guide for additional information about LOGPRT.) The problem could be a hardware problem.

Make sure that the CREATK template is the same size as the FD in COBOL or the buffer size in FORTRAN.

Code	Explanation
20	An error was encountered while writing a record or index block.
21	An error was encountered while reading a data record or index block. When using FORTRAN, it is necessary to use K$GETU rather than hard-coded file units so that MIDASPLUS can monitor the file units that are open.
22	A file system error was encountered while attempting to get a file unit or the internal file unit table is full. When using FORTRAN, it is necessary to use K$GETU rather than hard-coded file units so that MIDASPLUS can monitor the file units that are open.
23	The unit is not open as a segment directory.
28	You attempted to write to a read-only file.

Programming Error Codes

Error codes in the 30 range are usually programming errors: for example, reading past the end of a file. Check your program if you receive an error code in this range.

Code	Explanation
30	You did not ask for the array to be returned when it must be returned. Set FL$RET in flags on the call.
31	The array must be supplied but was not. Set the flag FL$USE.
32	You supplied a bad length (for example an invalid record length) or the index supplied is invalid.
33	You supplied an invalid array.
34	The use of NEXT$ is not allowed in direct access files.
35	You cannot do an indexed add to a direct access file.
36	You set FL$USE in flags but the current array involved a different index from the one that you supplied in this call.

Internal Error Codes

Error codes in the 40 range are usually internal errors. If you receive a 40 or 41 error message, run MPLUSCLUP -ALL from the supervisor terminal. See Chapter 12, CLEANING UP A MIDASPLUS FILE. If the problem persists, reshare MIDASPLUS.

If you receive a 42-46 error, there are corrupted pointers in the file. Run MPACK with the DATA option. See Chapter 15, PACKING A MIDASPLUS FILE. If the problem persists run MDUMP and KBUILD to restructure the file.

Code	Explanation
40	Fatal internal error.
41	Timeout occurred while waiting for buffers.
42	There is no offspring pointer or no next block found. The index tree is corrupted.
44	You attempted to access an indexed file as direct access or the file is direct access and the entry was not found.
45	Not a valid index block.
46	Not a valid data record.

Additional Miscellaneous Error Codes

Code	Explanation
51	Invalid index pointer in index entry. For example, segment number is 0.
71	An error occurred while attempting to delete an entry from a direct access file.
85	The index or data subfiles are full. Use the EXTEND option of CREATK to extend the subfile. Increase the subfile in a multiple of 512,000. Use MPACK with the DATA option on the file after the EXTEND to restructure the indexes and the data subfiles.
92	Network error.

10001	You supplied an invalid parameter on an OPEN or CLOSE.
10002	The MIDASPLUS internal tables are full.
10003	Not a segment directory.
10004	Fatal internal error. Contact Prime Customer Service.
10005	Error opening remote file.
10006	Maximum number of configured users exceeded.
10007	File in use by MIDASPLUS.
10008	File in use by MIDASPLUS utility.
10009	Exceeding maximum number of remote files in CLOSE time.

COBOL STATUS CODES

The following section lists the COBOL status codes and the equivalent MIDASPLUS error codes and states whether the status codes appear with INDEXED (I) and/or RELATIVE (R) files.

Status Code	MIDASPLUS Code	File Type	Interpretation
00			Successful completion of the operation.
10	7	I, R	The end of file was reached on a READ operation. The file pointer is positioned past the logical end of the file.
22	12	I, R	An attempt was made to perform a WRITE or REWRITE that would create a duplicate primary key entry. Duplicate primary key values are illegal.
23	7	I, R	The record was not found on an unsuccessful key search. There is no record in the file with this key value.
30	19/20	I, R	An error parity such as quota exceeded or disk full.
90	10	I, R	The record is already locked. Another user or process has already locked this record for update.
91	11	I, R	The record is not locked. A REWRITE operation was attempted without first locking the record with a READ operation.
92	12	I	An attempt was made to add a duplicate secondary key value to a secondary index subfile that does not permit duplicates.
93	30-33	I	The indexes referred to in the program do not match those defined during template creation.
94	13	I, R	A MIDASPLUS concurrency error. The command attempted to operate on a record that another user just deleted.

95	32	I, R	A record length was supplied for the file that does not match the record size assigned to the file during template creation.
96	20/21	R	A record number was supplied larger than the number that CREATK allocated.
98	35	R	An attempt was made to do an indexed add to a direct access file. Entries cannot be added to a RELATIVE file even if it is opened for INDEX access.
99		I, R	Any MIDASPLUS system error (greater than 40) that cannot be handled at the program level.

C

PRIMOS Error Messages

This appendix defines PRIMOS error messages and codes. See the Subroutine Reference Guide for additional information.

```
/* ERRD.INS.PLP, PRIMOS>INSERT, PRIMOS GROUP, 02/13/85
   MNEMONIC CODES FOR FILE SYSTEM (PL1)
   Copyright (c) 1982, Prime Computer, Inc., Natick, MA 01760 */

     /*************************************************************/
     /*                                                           */
     /*                                                           */
     /*      CODE DEFINITIONS                                     */
     /*                                                           */
     /*                                                           */
     E$EOF  BY  00001, /* END OF FILE                  PE       */
     E$BOF  BY  00002, /* BEGINNING OF FILE            PG       */
     E$UNOP BY  00003, /* UNIT NOT OPEN                PD,SD    */
     E$UIUS BY  00004, /* UNIT IN USE                  SI       */
     E$FIUS BY  00005, /* FILE IN USE                  SI       */
     E$BPAR BY  00006, /* BAD PARAMETER                SA       */
     E$NATT BY  00007, /* NO UFD ATTACHED              SL,AL    */
     E$FDFL BY  00008, /* UFD FULL                     SK       */
     E$DKFL BY  00009, /* DISK FULL                    DJ       */
     e$disk_full
            by      9, /* alias to E$DKFL                       */
     E$NRIT BY  00010, /* NO RIGHT                     SX       */
     E$FDEL BY  00011, /* FILE OPEN ON DELETE          SD       */
     E$NTUD BY  00012, /* NOT A UFD                    AR       */
     E$NTSD BY  00013, /* NOT A SEGDIR                 --       */
     E$DIRE BY  00014, /* IS A DIRECTORY               --       */
```

```
E$FNTF BY  00015, /* (FILE) NOT FOUND                SH,AH    */
E$FNTS BY  00016, /* (FILE) NOT FOUND IN SEGDIR      SQ       */
E$BNAM BY  00017, /* ILLEGAL NAME                    CA       */
E$EXST BY  00018, /* ALREADY EXISTS                  CZ       */
E$DNTE BY  00019, /* DIRECTORY NOT EMPTY             --       */
E$SHUT BY  00020, /* BAD SHUTDN (FAM ONLY)           BS       */
E$DISK BY  00021, /* DISK I/O ERROR                  WB       */
E$BDAM BY  00022, /* BAD DAM FILE (FAM ONLY)         SS       */
E$PTRM BY  00023, /* PTR MISMATCH (FAM ONLY)         PC,DC,AC */
e$rec_hdr_ptr_mismatch
       by     23, /* alias to E$PTRM                          */
E$BPAS BY  00024, /* BAD PASSWORD (FAM ONLY)         AN       */
E$BCOD BY  00025, /* BAD CODE IN ERRVEC              --       */
E$BTRN BY  00026, /* BAD TRUNCATE OF SEGDIR          --       */
E$OLDP BY  00027, /* OLD PARTITION                   --       */
E$BKEY BY  00028, /* BAD KEY                         --       */
E$BUNT BY  00029, /* BAD UNIT NUMBER                 --       */
E$BSUN BY  00030, /* BAD SEGDIR UNIT                 SA       */
E$SUNO BY  00031, /* SEGDIR UNIT NOT OPEN            --       */
E$NMLG BY  00032, /* NAME TOO LONG                   --       */
E$SDER BY  00033, /* SEGDIR ERROR                    SQ       */
E$BUFD BY  00034, /* BAD UFD                         --       */
E$BFTS BY  00035, /* BUFFER TOO SMALL                --       */
E$FITB BY  00036, /* FILE TOO BIG                    --       */
E$NULL BY  00037, /* (NULL MESSAGE)                  --       */
E$IREM BY  00038, /* ILL REMOTE REF                  --       */
E$DVIU BY  00039, /* DEVICE IN USE                   --       */
E$RLDN BY  00040, /* REMOTE LINE DOWN                --       */
E$FUIU BY  00041, /* ALL REMOTE UNITS IN USE         --       */
E$DNS  BY  00042, /* DEVICE NOT STARTED              --       */
E$TMUL BY  00043, /* TOO MANY UFD LEVELS             --       */
E$FBST BY  00044, /* FAM - BAD STARTUP               --       */
E$BSGN BY  00045, /* BAD SEGMENT NUMBER              --       */
E$FIFC BY  00046, /* INVALID FAM FUNCTION CODE       --       */
E$TMRU BY  00047, /* MAX REMOTE USERS EXCEEDED       --       */
E$NASS BY  00048, /* DEVICE NOT ASSIGNED             --       */
E$BFSV BY  00049, /* BAD FAM SVC                     --       */
E$SEMO BY  00050, /* SEM OVERFLOW                    --       */
E$NTIM BY  00051, /* NO TIMER                        --       */
E$FABT BY  00052, /* FAM ABORT                       --       */
E$FONC BY  00053, /* FAM OP NOT COMPLETE             --       */
E$NPHA BY  00054, /* NO PHANTOMS AVAILABLE           -        */
E$ROOM BY  00055, /* NO ROOM                         --       */
E$WTPR BY  00056, /* DISK WRITE-PROTECTED            JF       */
E$ITRE BY  00057, /* ILLEGAL TREENAME                FE       */
E$FAMU BY  00058, /* FAM IN USE                      --       */
E$TMUS BY  00059, /* MAX USERS EXCEEDED              --       */
E$NCOM BY  00060, /* NULL_COMLINE                    --       */
E$NFLT BY  00061, /* NO_FAULT_FR                     --       */
E$STKF BY  00062, /* BAD STACK FORMAT                --       */
E$STKS BY  00063, /* BAD STACK ON SIGNAL             --       */
E$NCON BY  00064, /* NO ON UNIT FOR CONDITION        --       */
E$CRWL BY  00065, /* BAD CRAWLOUT                    --       */
E$CROV BY  00066, /* STACK OVFLO DURING CRAWLOUT     --       */
```

```
E$CRUN BY  00067, /* CRAWLOUT UNWIND FAIL              --       */
E$CMND BY  00068, /* BAD COMMAND FORMAT                --       */
E$RCHR BY  00069, /* RESERVED CHARACTER                --       */
E$NEXP BY  00070, /* CANNOT EXIT TO COMMAND PROC       --       */
E$BARG BY  00071, /* BAD COMMAND ARG                   --       */
E$CSOV BY  00072, /* CONC STACK OVERFLOW               --       */
E$NOSG BY  00073, /* SEGMENT DOES NOT EXIST            --       */
E$TRCL BY  00074, /* TRUNCATED COMMAND LINE            --       */
E$NDMC BY  00075, /* NO SMLC DMC CHANNELS              --       */
E$DNAV BY  00076, /* DEVICE NOT AVAILABLE              DPTX     */
E$DATT BY  00077, /* DEVICE NOT ATTACHED               --       */
E$BDAT BY  00078, /* BAD DATA                          --       */
E$BLEN BY  00079, /* BAD LENGTH                        --       */
E$BDEV BY  00080, /* BAD DEVICE NUMBER                 --       */
E$QLEX BY  00081, /* QUEUE LENGTH EXCEEDED             --       */
E$NBUF BY  00082, /* NO BUFFER SPACE                   --       */
E$INWT BY  00083, /* INPUT WAITING                     --       */
E$NINP BY  00084, /* NO INPUT AVAILABLE                --       */
E$DFD  BY  00085, /* DEVICE FORCIBLY DETACHED          --       */
E$DNC  BY  00086, /* DPTX NOT CONFIGURED               --       */
E$SICM BY  00087, /* ILLEGAL 3270 COMMAND              --       */
E$SBCF BY  00088, /* BAD 'FROM' DEVICE                 --       */
E$VKBL BY  00089, /* KBD LOCKED                        --       */
E$VIA  BY  00090, /* INVALID AID BYTE                  --       */
E$VICA BY  00091, /* INVALID CURSOR ADDRESS            --       */
E$VIF  BY  00092, /* INVALID FIELD                     --       */
E$VFR  BY  00093, /* FIELD REQUIRED                    --       */
E$VFP  BY  00094, /* FIELD PROHIBITED                  --       */
E$VPFC BY  00095, /* PROTECTED FIELD CHECK             --       */
E$VNFC BY  00096, /* NUMERIC FIELD CHECK               --       */
E$VPEF BY  00097, /* PAST END OF FIELD                 --       */
E$VIRC BY  00098, /* INVALID READ MOD CHAR             --       */
E$IVCM BY  00099, /* INVALID COMMAND                   --       */
E$DNCT BY  00100, /* DEVICE NOT CONNECTED              --       */
E$BNWD BY  00101, /* BAD NO. OF WORDS                  --       */
E$SGIU BY  00102, /* SEGMENT IN USE                    --       */
E$NESG BY  00103, /* NOT ENOUGH SEGMENTS (VINIT$)      --       */
E$SDUP BY  00104, /* DUPLICATE SEGMENTS (VINIT$)       --       */
E$IVWN BY  00105, /* INVALID WINDOW NUMBER             --       */
E$WAIN BY  00106, /* WINDOW ALREADY INITIATED          --       */
E$NMVS BY  00107, /* NO MORE VMFA SEGMENTS             --       */
E$NMTS BY  00108, /* NO MORE TEMP SEGMENTS             --       */
E$NDAM BY  00109, /* NOT A DAM FILE                    --       */
E$NOVA BY  00110, /* NOT OPEN FOR VMFA                 --       */
E$NECS BY  00111, /* NOT ENOUGH CONTIGUOUS SEGMENTS              */
E$NRCV BY  00112, /* REQUIRES RECEIVE ENABLED          --       */
E$UNRV BY  00113, /* USER NOT RECEIVING NOW            --       */
E$UBSY BY  00114, /* USER BUSY, PLEASE WAIT            --       */
E$UDEF BY  00115, /* USER UNABLE TO RECEIVE MESSAGES             */
E$UADR BY  00116, /* UNKNOWN ADDRESSEE                 --       */
E$PRTL BY  00117, /* OPERATION PARTIALLY BLOCKED       --       */
E$NSUC BY  00118, /* OPERATION UNSUCCESSFUL            --       */
E$NROB BY  00119, /* NO ROOM IN OUTPUT BUFFER          --       */
E$NETE BY  00120, /* NETWORK ERROR ENCOUNTERED         --       */
```

```
        E$SHDN BY  00121, /* DISK HAS BEEN SHUT DOWN        FS        */
        E$UNOD BY  00122, /* UNKNOWN NODE NAME (PRIMENET)             */
        E$NDAT BY  00123, /* NO DATA FOUND                  --        */
        E$ENQD BY  00124, /* ENQUED ONLY                    --        */
        E$PHNA BY  00125, /* PROTOCOL HANDLER NOT AVAIL     DPTX      */
        E$IWST BY  00126, /* E$INWT ENABLED BY CONFIG       DPTX      */
        E$BKFP BY  00127, /* BAD KEY FOR THIS PROTOCOL      DPTX      */
        E$BPRH BY  00128, /* BAD PROTOCOL HANDLER (TAT)     DPTX      */
        E$ABTI BY  00129, /* I/O ABORT IN PROGRESS          DPTX      */
        E$ILFF BY  00130, /* ILLEGAL DPTX FILE FORMAT       DPTX      */
        E$TMED BY  00131, /* TOO MANY EMULATE DEVICES       DPTX      */
        E$DANC BY  00132, /* DPTX ALREADY CONFIGURED        DPTX      */
        E$NENB BY  00133, /* REMOTE MODE NOT ENABLED         NPX      */
        E$NSLA BY  00134, /* NO NPX SLAVE AVAILABLE          ---      */
        E$PNTF BY  00135, /* PROCEDURE NOT FOUND            R$CALL    */
        E$SVAL BY  00136, /* SLAVE VALIDATION ERROR         R$CALL    */
        E$IEDI BY  00137, /* I/O error or device interrupt (GPPI)     */
        E$WMST BY  00138, /* Warm start happened (GPPI)               */
        E$DNSK BY  00139, /* A pio instruction did not skip (GPPI)    */
        E$RSNU BY  00140, /* REMOTE SYSTEM NOT UP           R$CALL    */
        E$S18E BY  00141,
/*                                                                    */
/*      New error codes for REV 19 begin here:                        */
/*                                                                    */
        E$NFQB BY  00142, /* NO FREE QUOTA BLOCKS           --        */
        E$MXQB BY  00143, /* MAXIMUM QUOTA EXCEEDED         --        */
        e$max_quota_exceeded
               by    143, /* alias to E$MXQB                          */
        E$NOQD BY  00144, /* NOT A QUOTA DISK (RUN VFIXRAT)           */
        E$QEXC BY  00145, /* SETTING QUOTA BELOW EXISTING USAGE       */
        E$IMFD BY  00146, /* Operation illegal on MFD                 */
        E$NACL BY  00147, /* Not an ACL directory                     */
        E$PNAC BY  00148, /* Parent not an ACL directory              */
        E$NTFD BY  00149, /* Not a file or directory                  */
        E$IACL BY  00150, /* Entry is an ACL                          */
        E$NCAT BY  00151, /* Not an access category                   */
        E$LRNA BY  00152, /* Like reference not available             */
        E$CPMF BY  00153, /* Category protects MFD                    */
        E$ACBG BY  00154, /* ACL too big                              */
        E$ACNF BY  00155, /* Access category not found                */
        E$LRNF BY  00156, /* Like reference not found                 */
        E$BACL BY  00157, /* BAD ACL                                  */
        E$BVER BY  00158, /* BAD VERSION                              */
        E$NINF BY  00159, /* NO INFORMATION                           */
        E$CATF BY  00160, /* Access category found (Ac$rvt)           */
        E$ADRF BY  00161, /* ACL directory found (Ac$rvt)             */
        E$NVAL BY  00162, /* Validation error (nlogin)                */
        E$LOGO BY  00163, /* Logout (code for fatal$)                 */
        E$NUTP BY  00164, /* No unit table available. (PHANT$)        */
        E$UTAR BY  00165, /* Unit table already returned. (UTDALC)    */
        E$UNIU BY  00166, /* Unit table not in use. (RTUTBL)          */
        E$NFUT BY  00167, /* No free unit table. (GTUTBL)             */
        E$UAHU BY  00168, /* User already has unit table. (UTALOC)    */
        E$PANF BY  00169, /* Priority ACL not found.                  */
```

```
E$MISA BY  00170, /* Missing argument to command.                    */
E$SCCM BY  00171, /* Supervisor terminal command only.               */
E$BRPA BY  00172, /* Bad remote password              R$CALL         */
E$DTNS BY  00173, /* Date and time not set yet.                      */
E$SPND BY  00174, /* REMOTE PROCEDURE CALL STILL PENDING             */
E$BCFG BY  00175, /* NETWORK CONFIGURATION MISMATCH                  */
E$BMOD BY  00176, /* Illegal access mode  (AC$SET)                   */
E$BID  BY  00177, /* Illegal identifier   (AC$SET)                   */
E$ST19 BY  00178, /* Operation illegal on pre-19 disk                */
E$CTPR BY  00179, /* Object is category-protected (Ac$chg)           */
E$DFPR BY  00180, /* Object is default-protected (Ac$chg)            */
E$DLPR BY  00181, /* File is delete-protected (Fil$dl)               */
E$BLUE BY  00182, /* Bad LUBTL entry                 (F$IO)         */
E$NDFD BY  00183, /* No driver for device            (F$IO)         */
E$WFT  BY  00184, /* Wrong file type                 (F$IO)         */
E$FDMM BY  00185, /* Format/data mismatch            (F$IO)         */
E$FER  BY  00186, /* Bad format                      (F$IO)         */
E$BDV  BY  00187, /* Bad dope vector                 (F$IO)         */
E$BFOV BY  00188, /* F$IOBF overflow                 (F$IO)         */
E$NFAS BY  00189, /* Top-level dir not found or inaccessible*/
E$APND BY  00190, /* Asynchronous procedure still pending            */
E$BVCC BY  00191, /* Bad virtual circuit clearing                    */
E$RESF BY  00192, /* Improper access to a restricted file            */
E$MNPX BY  00193, /* Illegal multiple hops in NPX.                   */
E$SYNT BY  00194, /* SYNTanx error                                   */
E$USTR BY  00195, /* Unterminated STRing                             */
E$WNS  BY  00196, /* Wrong Number of Subscripts                      */
E$IREQ BY  00197, /* Integer REQuired                                */
E$VNG  BY  00198, /* Variable Not in namelist Group                  */
E$SOR  BY  00199, /* Subscript Out of Range                          */
E$TMVV BY  00200, /* Too Many Values for Variable                    */
E$ESV  BY  00201, /* Expected String Value                           */
E$VABS BY  00202, /* Variable Array Bounds or Size                   */
E$BCLC BY  00203, /* Bad Compiler Library Call                       */
E$NSB  BY  00204, /* NSB tape was detected                           */
E$WSLV BY  00205, /* Slave's ID mismatch                             */
E$VCGC BY  00206, /* The virtual circuit got cleared.                */
E$MSLV BY  00207, /* Exceeds max number of slaves per user           */
E$IDNF BY  00208, /*  Slave's ID not found                           */
E$NACC BY  00209, /* Not accessible                                  */
E$UDMA BY  00210, /* Not Enough DMA channels                         */
E$UDMC BY  00211, /* Not Enough DMC channels                         */
E$BLEF BY  00212, /* Bad tape record length and EOF                  */
E$BLET BY  00213, /* Bad tape record length and EOT                  */
E$ALSZ BY  00214, /* Allocate request too small                      */
E$FRER BY  00215, /* Free request with invalid pointer               */
E$HPER BY  00216, /* User storage heap is corrupted                  */
E$EPFT BY  00217, /* Invalid EPF type                                */
E$EPFS BY  00218, /* Invalid EPF search type                         */
E$ILTD BY  00219, /* Invalid EPF LTD linkage descriptor              */
E$ILTE BY  00220, /* Invlaid EPF LTE linkage descriptor              */
E$ECEB BY  00221, /* Exceeding command environment breadth           */
E$EPFL BY  00222, /* EPF file exceeds file size limit                */
E$NTA  BY  00223, /* EPF file not active for this user               */
```

```
        E$SWPS BY  00224, /* EPF file suspended within program session *
/
        E$SWPR BY  00225, /* EPF file suspended within this process */
        E$ADCM BY  00226, /* System administrator command ONLY      */
        E$UAFU BY  00227, /* Unable to allocate file unit           */
        e$unable_to_allocate_file_unit
               by  00227, /* alias to E$UAFU                        */
        E$FIDC BY  00228, /* File inconsistent data count           */
        e$file_inconsistent_data_count
               by  00228,  /* alias to e$fidc                       */
        e$indl by  00229,  /* alias to e$insufficient_dam_level     */
        e$insufficient_dam_levels
               by  00229, /* Not enough dam index levels as needed  */
        e$peof by  00230, /* alias to e$past_EOF                    */
        e$past_EOF
               by  00230, /* Past End Of File                       */
        E$N231 by  00231, /* Error code 231.                        */
        E$N232 by  00232, /* Error code 232.                        */
        E$N233 by  00233, /* Error code 233.                        */
        E$N234 by  00234, /* Error code 234.                        */
        E$N235 by  00235, /* Error code 235.                        */
        E$N236 by  00236, /* Error code 236.                        */
        E$N237 by  00237, /* Error code 237.                        */
        E$N238 by  00238, /* Error code 238.                        */
        E$RSHD by  00239, /* Remote disk has been shut down.        */
        E$N240 by  00240, /* No paging device defined.              */
        e$nrfc by  00241, /* Specified reverse flow control on.     */
        e$N242 by  00242, /* Error code 242                         */
        e$N243 by  00243, /* Error code 243                         */
        e$N244 by  00244, /* Error code 244                         */
        e$aele by  00245, /* Attempt to execute non-executable library *
/
        E$LAST BY  00245; /* THIS ***MUST*** BE. LAST       --      */
        /*                                                         */
        /*  The value of E$LAST must equal the last error code.    */
        /*                                                         */
        /*********************************************************/
```

D

Using Prime Customer Service

This appendix lists the information that Prime's Customer Service Department needs in order to help you diagnose errors with MIDASPLUS. Please consult Appendix B, ERROR MESSAGES, and Appendix C, PRIMOS ERROR MESSAGES, before calling Customer Service. If it is necessary to contact Customer Service, please be prepared to answer the following questions:

- What revision of PRIMOS are you using?
- Is there anything non-standard (for example, upgrades or patches) about your system?
- What revision, including fix release, of MIDASPLUS are you using?
- How did you respond to the questions in the CREATK dialog?
- How did you build your file? (KBUILD, ADD1$, or offline routines?)
- Is the application a single-user or a multiple-user application?
- Is the file local or remote?
- What language is the application written in?
- What MIDASPLUS utilities have been used on the file during the last successful run?

E

Concurrency Issues

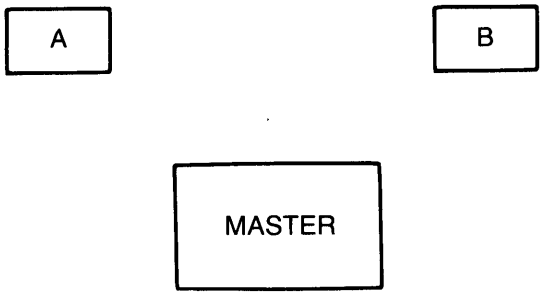
This appendix discusses the special considerations for writing application programs that allow concurrent access to data records within a MIDASPLUS file. These considerations include the necessary programming procedures that the application programmer must code to handle concurrency problems when reading, updating, and deleting records from a MIDSAPLUS file that is open by multiple updaters.

LOCKED RECORDS

Locking a record during an update process ensures that only one user can update a record at one time. Otherwise, updates are inconsistent and data integrity is lost. It is the responsibility of the application programmer to code a procedure to handle locked record conditions.

Example:

Assume that two users (User-A and User-B) have a file (MASTER) open for update (reading and writing). The MASTER file contains customer records with Primary keys ranging from 1000 to 9999. The application program is written in COBOL. See Figure E-1.



Concurrency Issues
MASTER File

Figure E-1

When User-A and User-B open the file for update (INPUT-OUTPUT) and both users attempt to access the same record (Customer-1002) the following occurs:

Time	User-A	User-B	Comments
1	Read Customer-1002	---	User-A retrieves Customer-1002 and locks the record.
2	Process Data	Read Customer-1002	User-B gets an error code 10 (or COBOL 90) indicating that the record is locked. User-B cannot read the record at this time and must retry.
3	Rewrite Customer-1002	---	After the update process within MIDASPLUS is completed, the record is released.
4		Read Customer-1002	Since User-A has released the record lock, User-B is now able to read and update the record. User-B now has the the record locked.

It is important to note that if User-B had opened the file for read only, then User-B could have read the locked record.

When you use the MIDASPLUS interface to COBOL, the following sequence of MIDASPLUS calls are executed.

CALL LOCK$	Finds and locks a record. The routine LOCK$ cannot access a record that is already locked.
CALL UPDAT$	Updates and unlocks the record. This routine can also be used to unlock a record without an actual update.

It is up to the programmer to code a routine within the application to handle locked record conditions. For example:

```
    READ MASTER-FILE INTO CUSTOMER-RECORD
                  INVALID KEY PERFORM ERROR-RTN THRU
                                      ERROR-RTN-EXIT.
       :
       :
       :
ERROR-RTN.
   IF FILE-STATUS = 90 CALL 'SLEEP$' USING SLEEP-TIME
      ELSE .....
ERROR-RTN-EXIT.
   EXIT.
```

DELETED RECORDS

Locked records can be deleted by another user. When writing applications to delete and update records concurrently, make sure that the record is read and locked before deleting it. This ensures that any user attempting to update the record will get a locked record status code of 10 or COBOL 90. If the record is not read prior to the delete, the following may occur assuming that both User-A and User-B opened the file for input and output:

Time	User-A	User-B	Comments
1		Read record Customer-1002	User-B reads and locks the record.
2	Deletes record Customer-1002 without first retrieving the record to lock it.		User-A deletes the locked record.
3		Rewrite record Customer-1002	User-B is unable to rewrite the record and receives a MIDASPLUS status code of 11 (COBOL 91).

A status code of 13 (COBOL 94) may occur when one user deletes a record that another user is attempting to access. The second user may lose his/her position in the file which results in this concurrency error.

When any error condition occurs during an attempt to access a record, the file's position may become undefined. Therefore, it may be necessary for the program to reestablish the positioning the file before attempting another access.

COBOL SEQUENTIAL ACCESS

When sequentially accessing a MIDASPLUS file to update and delete records in a concurrent update mode, there are additional programming considerations that must be taken.

Consider the following program:

```
     SELECT MASTER-FILE ASSIGN TO PFMS
            ORGANIZATION IS INDEXED
            ACCESS MODE IS SEQUENTIAL (or DYNAMIC)
            :
            :
    FD MASTER-FILE
            :
            :
            :
         OPEN I-O MASTER-FILE
            :
         START MASTER-FILE.....
            :
      READ-RECORD-PARA.
            :
         READ MASTER-FILE NEXT RECORD INTO MASTER-RECORD
                      AT END GO TO EOJ-RTN.
            :
         REWRITE MASTER-RECORD INVALID KEY PERFORM ERROR-RTN.
            :
         GO TO READ-RECORD-PARA.
```

When multiple users are running the above program to access the same file, several unanticipated error conditions may occur.

For example, the syntax specification for this particular format of the READ statement does not allow for an INVALID KEY clause. If another user already has a record locked while this READ statement is attempting to retrieve a record, the program will abort with a MIDASPLUS status code 10 or COBOL 90. In order to trap the locked record status, the application programmer must code a DECLARATIVES section to enable the program to test the file status code that MIDASPLUS returns.

Since this is a sequential read, any error condition that MIDASPLUS returns results in an undefined position in the file. It is imperative that the application programmer ensure that the program reposition

itself in the file using the START statement. In the above example, the program should return to the START statement when any recoverable error condition is encountered during a read-next operation. Accessing a record by full primary key will also reestablish the position in the file.

Example:

```
PROCEDURE DIVISION.
    DECLARATIVES SECTION.
    :
    :
MAIN-PARA.
    OPEN I-O MASTER-FILE.
    :
START-PARA.
    START MASTER-FILE.......
    :
READ-RECORD-PARA.
    :
    READ MASTER-FILE NEXT RECORD AT END GO TO EOJ-RTN.
    IF FILE-STATUS = 90 GO TO START-PARA.
    :
    REWRITE MASTER-RECORD INVALID KEY PERFORM ERROR-RTN.
GO TO READ-RECORD-PARA.
```

HARD-CODED FILE UNITS

Since MIDASPLUS manages file units for each user, the use of hard-coded file units can cause MIDASPLUS files to become corrupted beyond repair. For example, MIDASPLUS maintains a table that contains entries for each file unit with corresponding file names or subfile numbers. Since a MIDASPLUS file contains many subfiles, you are not aware that MIDASPLUS is opening subfiles on other file units. If MIDASPLUS already has a subfile opened on file unit 122, and a user program attempts to open another file on unit 122 using TSRC$$, the subfile will be closed without notifying MIDASPLUS. MIDASPLUS will access information on file unit 122 which is incorrect and subsequently damage the file open on unit 122 (which may be another MIDASPLUS file). In addition, MIDASPLUS may use information in the subfile on unit 122 to access or update information in another subfile.

Besides possibly corrupting the MIDASPLUS file, hard-coded file units may cause the following error conditions to occur:

```
NOT A VALID MIDASPLUS FILE
UNIT NOT OPEN
Error code 21.
```

MIDASPLUS also uses the internal file unit table to efficiently manage units assigned to MIDASPLUS files. If the user has many MIDASPLUS files open at one time and the pool of available file units for MIDASPLUS has been exhausted, MIDASPLUS closes a file unit and uses that file unit to open another MIDASPLUS file. When the program requires use of a file unit that MIDASPLUS has closed and assigned to another file, MIDASPLUS may reopen the file on another previously assigned file unit. Managing the file units is dependent on the number of file units configured in the MIDASPLUS configuration file. See Appendix I, FILE UNIT MANAGEMENT, for additional information about hard-coded files.

CONCURRENCY RULES

Always be very careful when coding programs that will be used by multiple users to access the same files. Some basic rules are:

1. Always check error codes and design the program to take appropriate action when recoverable error conditions are encountered.

2. Be careful when deleting records in a concurrent access mode of operation since another user can delete a locked record. When possible, design the application to delete records in batch or exclusive update mode.

3. Because the CBL compiler does not allow the INVALID KEY clause on a READ NEXT statement, use a DECLARATIVE section to trap error conditions when sequentially accessing a file in multi-user mode.

4. When an error occurs on a sequential read (READ NEXT), the program's position in the file becomes undefined and a subsequent READ NEXT generates an end-of-file error.

5. When using the MIDASPLUS call interface (ADD1$, FIND$, etc.) always use OPENM$ and CLOSM$ to open and close files.

6. When using the MIDASPLUS call interface, never use hard-coded file units. Always use K$GETU to ensure that MIDASPLUS properly manages file units. Use K$GETU regardless of file type.

7. When using either the COBOL interface or the library routines, do not open a file multiple times in a program. Always make sure that files are closed when the program is terminated.

8. It is also very important to make sure MPLUSCLUP is invoked when a program terminates abnormally. This ensures that MIDASPLUS closes all file units and releases all file locks held by the users.

F

Other MIDASPLUS Offline Routines

ERROPN and KX$TIM are only for FORTRAN users who use offline routines. This appendix discusses ERROPN and KX$TIM.

ERROPN

ERROPN is a routine used to open a logging file to record errors and milestone statistics. KBUILD uses ERROPN to open and name an error/logfile. The calling sequence for ERROPN is:

```
CALL ERROPN (funit)
```

Funit is the INT*2 file unit on which the error/logging file will be opened. It is returned as 0 if no file was opened.

Using the Routine

ERROPN prompts you with

```
ENTER LOG/ERROR FILE NAME:
```

to ask you for the error/logging file. If you press the RETURN key or blank line, funit is set to 0 and ERROPN returns you to your program. If you enter a valid pathname, a new SAM file is created if necessary, 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, you are asked to enter another pathname. The truncate operation makes 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.

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 you may have closed the file in the meantime. Errors occurring from attempts to write to this file are interpreted as a sign that the file has been closed and are ignored.

KX$TIM

KX$TIM is a user-callable routine that displays milestone statistics (including CPU, disk, and wall clock time elapsed since the last milestone) for the offline file-building routines PRIBLD, SECBLD, and BILD$R. These milestones are displayed at the terminal. If ERROPN was called to open an error/log file, these statistics are written to the error/log file. KBUILD uses KX$TIM to generate milestones.

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 0 and -1 make it possible to generate headers and so forth. See Using the Routine below.
optmsg	User-supplied if desired (INT*2). If supplied, the length of the optional message, in words, must be passed in msglen.

msglen	The length of the optional optmsg, in characters (INT*2). Set it to 0 if there is no optional message.

Using the Routine

If there is an optional message, it is printed to the terminal and to the optional error/logging file. 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 0, all counters are initialized to 0 and a header is printed out before the milestone line. By using a numrec value of -1, a milestone of 0 without a header or initialization can be generated.

G

The Call Interface with C

This appendix demonstrates how C programmers can access a MIDASPLUS file through subroutine calls in the MIDASPLUS library. You should have an in-depth understanding of MIDASPLUS before attempting to code in this manner.

When using the callable interface, always use OPENM$ and CLOSM$ to open and close MIDASPLUS files rather than using PRIMOS file routines such as SRCH$$ and TSRCH$$. The program should never contain hard-coded file units when you are opening a MIDASPLUS file. Always use K$GETU to allow PRIMOS to select the next available file unit, regardless of the file type.

C does not support the alternate return argument. When handling errors in C programs, use a 0 for the alternate return argument and check the communications array after the call.

MIDASPLUS routines that do not support a communications array argument (for example, PRIBLD) terminate the program with an error message if you use 0 instead of an alternate error return. MIDASPLUS routines that support the communications array and classify errors as fatal and nonfatal (for example ADD1$) terminate if a fatal error occurs and you use 0 instead of an alternate return.

To handle the above error situation from C, use an interlude routine written in another language that provides the alternate error. The following is an example of an interlude routine written in FORTRAN for ADD1$:

```
      SUBROUTINE ADD1$C(FUNIT,BUFFER,KEY,ARRAY,FLAGS,FATAL,INDEX,
     *     FILENO,PLENTH,KEYLNT)
      INTEGER FATAL
      CALL ADD1$(FUNIT,BUFFER,KEY,ARRAY,FLAGS,$1,INDEX,
     *     FILENO,PLENTH,KEYLNT)
      FATAL = 0
      RETURN
    1 FATAL = 1
      RETURN
      END
```

CALLABLE INTERFACE EXAMPLE

The following C program reads data from the MIDASPLUS file that was created in Chapter 2.

```
/* This is a C program that opens an indexed file,          */
/* reads a record, and displays it.                         */

/* IT MUST BE COMPILED WITH -NEWFORTRAN                     */


#include "syscom>parm.k.ins.cc"     /* MIDASPLUS flag values */

#include "syscom>keys.ins.cc"            /* Primos I/O keys */


#include <stdio.h>                    /*   needed for getchar */

main()
{
/* Data structures:  */
fortran closm$(), find$(), openm$();
short int funit, i, status, routine, buffer[43];
short int array[14];
char choice;
static struct thekey
        {char one[9];
        };   /*end struct*/
static struct thekey findkey;
```

```
/* START EXECUTION:  */

/*  Open file: */
    openm$((short) k$rdwr+k$getu, "bank", 4, funit, status);
    if (status != 0)
        abort();
/*  Ask for key to be entered from terminal:  */
    choice = 'Y';    /*  next while is repeated as long as choice is yes    */
    while ((choice == 'Y') | (choice == 'y'))
       {printf("ENTER KEY VALUE (9 NUMBERS): \n");
        i = 0;

        while (i <=8)
           {findkey.one[i] = getchar();
            i++;
            }  /* end while  */

/* Read and display sequential record: */
        find$(funit,
              buffer,
              findkey,
              array,
              (short) FL$RET,
              (long) 0,                          /* ALTRTN -- no use in C but
                                                          must be long */
              0,                                 /* search on primary key */
              0,                                 /* obsolete for MIDASPLUS */
              0,                                 /* return all data */
              0);                                /* full key */

/* check error code in array:  */

        if (array[0] == 0);                      /* do nothing, 0 is normal  */
        else
            if (array[0] == 7)                 /* key not found   */
                printf("THERE IS NO RECORD WITH THIS KEY\n");
            else
               {printf("ERROR -- ASK FOR HELP\n");
                abort();
                } /* end else */

/* display what is returned in buffer:  */
        printf("%s\n", buffer);
        printf("\n");
        printf("DO YOU WANT TO CONTINUE? Y or N:\n");
        i= 0;
        getchar();  /* throw away last CR */
        choice = getchar();
        getchar();  /* throw away last CR */
        }  /* end while for choice*/

/* Close file:  */
    closm$(funit, status);
    if (status == 0)
        printf ("NORMAL END OF RUN ");
    else
        printf ("STATUS IS", "%d\n", status);

}  /* end program */
```

```
OK, cc sample.cc -newfortran
[CC Rev. 21.0 Copyright (c) Prime Computer, Inc. 1986]
00 Errors and 00 Warnings detected in 85 lines and 646 include lines.
OK, bind
[BIND Rev. 21.0 Copyright (c) 1985, Prime Computer, Inc.]
: load sample
: li c_lib
: li mpluslb
BIND COMPLETE
: file
OK, resume sample
MIDASPLUS User initialized.
CLEANUP of 0 items complete.
ENTER KEY VALUE (9 NUMBERS):
189264289
189264289murray, paul              mc28374646123 orchard rd  manchester     nh03102

DO YOU WANT TO CONTINUE? Y or N:
y
ENTER KEY VALUE (9 NUMBERS):
276503889
276503889harper, anne              chk412389112 washington stnewton         ma02159

DO YOU WANT TO CONTINUE? Y or N:
n
NORMAL END OF RUN OK,
```

H

The Call Interface with Pascal

This appendix demonstrates how Pascal programmers can access a MIDASPLUS file through subroutine calls in the MIDASPLUS library. You should have an in-depth understanding of MIDASPLUS before attempting to code in this manner.

When using the callable interface, always use OPENM$ and CLOSM$ to open and close MIDASPLUS files rather than using PRIMOS file routines such as SRCH$$ and TSRCH$$. The program should never contain hard-coded file units when you are opening a MIDASPLUS file. Always use K$GETU to allow PRIMOS to select the next available file unit, regardless of the file type.

Pascal does not support the alternate return argument. When handling errors in Pascal programs, use a 0 for the alternate return argument and check the communications array after the call.

MIDASPLUS routines that do not support a communications array argument (for example, PRIBLD) terminate the program with an error message if you use 0 instead of an alternate error return. MIDASPLUS routines that support the communications array and classify errors as fatal and nonfatal (for example ADD1$) terminate if a fatal error occurs and you use 0 is instead of an alternate return.

To handle the above error situation from Pascal, use an interlude routine written in another language that provides the alternate error. The following is an example of an interlude routine written in FORTRAN for ADD1$:

```
      SUBROUTINE ADD1$P(FUNIT,BUFFER,KEY,ARRAY,FLAGS,FATAL,INDEX,
     *     FILENO,PLENTH,KEYLNT)
      INTEGER FATAL
      CALL ADD1$(FUNIT,BUFFER,KEY,ARRAY,FLAGS,$1,INDEX,
     *     FILENO,PLENTH,KEYLNT)
      FATAL = 0
      RETURN
    1 FATAL = 1
      RETURN
      END
```

CALLABLE INTERFACE EXAMPLE WITH PASCAL

The following Pascal program reads data from the MIDASPLUS file that was created in Chapter 2.

```
program customer(bank,input,output);

{This Pascal program opens a MIDASPLUS file, reads a record,
and displays it.}

const
  %include 'SYSCOM›PARM.K.INS.PASCAL';
  %include 'SYSCOM›KEYS.INS.PASCAL';

type
  chartype = packed array [1..88] of char;     {for the buffer in FIND$}
  inttype = array [1..14] of integer;     {for the communications array}
  string1 = packed array [1..4] of char;       {for pathname in OPENM$}
  string2 = packed array [1..9] of char;       {for key in FIND$}

var
  bank : text;
  funit, flags, altrtn, index, bufsiz, keysiz : integer;
  filno : integer;
  buffer : chartype;
  key : string2;
  array1 : inttype;
  pathname : string1;
  namlen : integer;
  k : integer;
  status : integer;
  primkey : string2;
  done : boolean;
  ans : char;
```

```
procedure FIND$(var funit : integer; var buffer : chartype;
                var key : string2; var array1 : inttype;
                var flags, altrtn, index, filno, bufsiz,
                keysiz : integer); extern;

procedure OPENM$(var k : integer; var pathname : string1;
                 var namlen : integer; var funit : integer;
                 var status : integer); extern;

procedure CLOSM$(var funit : integer; var status : integer); extern;

procedure print(buff : chartype);  {to write the buffer returned from f
ind}
var x : integer;
begin
  for x := 1 to 88 do
    begin
      write(buff[x]);
    end;
  writeln;
end;

begin
{First open the BANK file}
  k := K$GETU+K$RDWR;
  pathname := 'bank';
  namlen := 4;
  OPENM$(k, pathname, namlen, funit, status);

  done := false;

{Main loop}
  while not done do
    begin
      writeln('Enter the primary key (9 digits): ');
      readln(primkey);

{This section assigns values to the parameters of the procedure call FI
ND$}
    {funit - assigned by OPENM$}
    {buffer-empty character string}
      key := primkey;  {primary key of an existing entry}
      array1[1] := 0;  {flags sequential access}
      flags := 0;      {set it to zero}
      altrtn := 0;     {Pascal doesn't support alternate
                            return statements}
      index := 0;      {use the primary key}
      filno := 0;      {file no. obsolete, set to zero}
      bufsiz := 0;     {the complete buffer; length of the record}
      keysiz := 0;     {zero means use the whole key}
```

```
{Find the record with primary key PRIMKEY}
      FIND$(funit, buffer, key, arrayl, flags, altrtn, index,
            filno, bufsiz, keysiz);

      if arrayl[1] = 7 then
        writeln('There is no record with this key.')
      else
{Print the record to the terminal}
        print(buffer);
      writeln('Do you want to continue?');
      readln(ans);
      if (ans = 'n') or (ans = 'N') then
        done := true;
    end; {while}
{Close the file.}
  CLOSM$ (funit, status);
end.


OK, pascal customer
[PASCAL Rev. 21.0 Copyright (c) 1986 Prime Computer, Inc.]
0000 ERRORS [PASCAL Rev. 21.0]
OK, bind
[BIND Rev. 21.0 Copyright (c) 1985, Prime Computer, Inc.]
: load customer
: li paslib
: li mpluslb
: li
BIND COMPLETE
: file
OK, resume customer
Enter the primary key (9 digits):
276503889
276503889harper, anne              chk412389112 washington stnewton
    ma02159
Do you want to continue?
y
Enter the primary key (9 digits):
123456789
There is no record with this key.
Do you want to continue?
y
Enter the primary key (9 digits):
189264289
189264289murray, paul              mc28374646123 orchard rd  manchester
    nh03102
Do you want to continue?
n
OK,
```

I

File Unit Management

This appendix describes problems that occur when hard-coded file units are used. Since MIDASPLUS multiplexes file units for all open MIDASPLUS files, using hard-coded file units in application code will probably corrupt the MIDASPLUS internal file unit table and eventually damage MIDASPLUS files used by an application. The following conditions signal the corruption of the MIDASPLUS internal file unit table: MIDASPLUS error codes 21, 23, 40, and 45 and error messages such as UNIT NOT OPEN and NOT A SEGDIR.

MIDASPLUS FILE UNIT UTILIZATION

Since a MIDASPLUS file uses PRIMOS segment directories, a single MIDASPLUS file may be made up of many individual segment directory subfiles. Consequently, an application can require many PRIMOS file units.

For example, if a MIDASPLUS file has one primary key and five secondary keys, the file requires a minimum of 8 subfiles and 9 PRIMOS file units. One file unit is required for the segment directory and one file unit is required for each index subfile. These requirements result in 6 PRIMOS units for the indexes, and at least one file unit for the data subfile (subfile 185).

MIDASPLUS maintains an internal file unit table which is allocated on a per user basis. The purpose of this table is to enable MIDASPLUS to keep a record of the file units that are in use by active MIDASPLUS files and subfiles.

The funits parameter in the MPLUS.CONFIG file defines the per user file unit table size. IMIDASPLUS initializes the MIDASPLUS configuration at system startup or when MIDASPLUS is reshared. The following diagram shows the structure of a MIDASPLUS file unit table.

Index	xxunit 1=segdir	write 1=write	Remote 1=remote	file unit	Subfile # 0=segdir	Link

When an application calls MIDASPLUS to open a MIDASPLUS file with one primary and 5 secondary keys, the per-user file unit table looks like the following (assume that the funits parameter is set to 10):

Index	xxunit 1=segdir	write 1=write	Remote 1=remote	file unit	Subfile # 0=segdir	Link
128						
127						
126						
125	1	0	0	125	0	124
124	0	0	0	124	1	123
123	0	0	0	123	11	122
122	0	0	0	122	21	121
121	0	0	0	121	31	120
120	0	0	0	120	41	119
119	0	0	0	119	51	118
118	0	0	0	118	185	0

If the same application opens a second MIDASPLUS file with a primary key and 2 secondary keys, the MIDASPLUS file unit table is updated as follows:

Index	xxunit 1=segdir	write 1=write	Remote 1=remote	file unit	Subfile # 0=segdir	Link
128						
127						
126						
125	1	0	0	125	0	124
124	0	0	0	124	1	123
123	0	0	0	123	11	122
122	0	0	0	122	21	121
121	0	0	0	121	31	120
120	0	0	0	120	41	119
119	0	0	0	119	51	118
118	0	0	0	118	185	0
117	1	0	0	117	0	116
116	0	0	0	116	1	?

MIDASPLUS has now exhausted the number of file unit entries allowed by the funits configuration parameter. Since MIDASPLUS requires three additional file units, and MIDASPLUS will close units 124, 123, and 122. Note that MIDASPLUS never reuses the unit or the table entry that the segment directory occupies.

When MIDASPLUS opens the second file, the unit table appears as follows:

Index	xxunit 1=segdir	write 1=write	Remote 1=remote	file unit	Subfile # 0=segdir	Link
128						
127						
126						
125	1	0	0	125	0	121
124	0	0	0	124	11	123
123	0	0	0	123	21	122
122	0	0	0	122	185	0
121	0	0	0	121	31	120
120	0	0	0	120	41	119
119	0	0	0	119	51	118
118	0	0	0	118	185	0
117	1	0	0	117	0	116
116	0	0	0	116	1	124

By following the links for the second file, MIDASPLUS knows that this second file currently has units 117 (segment directory), 116, 124, 123, and 122. MIDASPLUS also knows that the first file is no longer using units 124, 123, and 122 since they are no longer on the chain associated with the segment directory at entry 125. This file is currently using units 125 (segment directory), 121, 120, 119, 118. The subfiles previously associated with entries 124, 123, and 122 have been closed and the corresponding unit number and table entries have been assigned to the second file.

When the application requires the use of subfiles 11, 21, and 31 of the first MIDASPLUS file, MIDASPLUS closes the appropriate units, removes them from the table, rebuilds the links, enters new units/subfiles into the table, and reestablishes the appropriate links.

POTENTIAL PROBLEMS

Consider the impact when the application in the previous example calls TSRC$$ to open a SAM file on hard-coded unit 120. (The first thing that TSRC$$ does before opening the file on unit 120 is to close unit 120.) Since MIDASPLUS was not notified that the file unit was closed (TSRC$$ is a PRIMOS routine and does not notify MIDASPLUS of any requests to open or close files), MIDASPLUS continued functioning as if subfile 41 was still using unit 120. If MIDASPLUS requires data from subfile 41, MIDASPLUS reads from the file that TSRC$$ opened on unit 120. As a result, MIDASPLUS recognizes inconsistent data structures and will probably abort the program. Possible errors include errors 21, 23, and 40. Before MIDASPLUS recognizes that the data structures are inconsistent, MIDASPLUS may use the data from unit 120 to update another MIDASPLUS file. If the application opens a SAM file on unit 120 for writing and MIDASPLUS attempts to update subfile 41, the actual updates are written to the file that was opened on unit 120. Since subfile 41 is not updated, inconsistencies occur in the MIDASPLUS file between the indexes and the data.

Additional problems occur if application programs do not check error codes on calls to SRCH$$, TSRCH$$, and OPENM$$. If PRIMOS returns an error condition on calls to open a file, and the error condition is ignored, the file unit parameter is not reset. The file unit parameter remains set to the previous value which will be used in subsequent calls to MIDASPLUS. As a result, MIDASPLUS may perform I/O using file units which are undefined and may be already in use by other MIDASPLUS files. In order to avoid problems, use the flag K$GETU instead of hard coding your file units and check the status code from any PRIMOS or MIDASPLUS routine that an application calls.

INDEX

Index

A

Access Control List, 16-3
  MIDASPLUS*, 16-4

Access mode statements,
  PL/I statement, 8-5

Access modes,
  COBOL, 6-18

Access Operations, 4-1

Access Path Statements, 6-10

Accessing,
  Locked Record, COBOL, 6-14
  MIDASPLUS, 1-2

ACL directory, 16-3

ADD,
  BASIC/VM, 7-7, 7-8
  CREATK, 2-12, 14-8, 14-9

ADD CREATK,
  function summary, 14-1

ADD1$ routine,
  arguments, 5-19, 5-20
  calling sequence, 5-18

ADD1$ routine (continued)
  flags, 5-21
  index values, 5-19
  keyed-index adds, 5-18

Adding data records,
  COBOL, 6-21
  FORTRAN, 5-21
  VRPG, 9-9

Adding secondary index entries,
    5-22

Administering MIDASPLUS,
  introduction, 16-1

ALTERNATE RECORD KEY, 6-4

Alternatives to KBUILD, 3-22

Argument listing,
  ADD1$, 5-19, 5-20
  CLOSM$, 5-14
  FIND$, 5-26
  NTFYM$, 5-15, 5-16
  OPENM$, 5-13

Arguments,
  optional, 5-11

Array format, 5-3
  direct access, 5-3
  keyed-index, 5-3

AT END clause, 6-11

Awaken, 13-5

B

BASIC/VM interface,
  ADD statement, 7-7, 7-8
  CLOSE statement, 7-4
  DEFINE FILE statement, 7-3
  error handling, 7-4
  introduction, 7-1
  language dependencies, 7-1, 7-2
  locking and unlocking records, 7-3
  MIDASERR, 7-5
  ON ERROR statement, 7-4
  POSITION statement, 7-5, 7-6
  READ statement, 7-9, 7-10
  REMOVE statement, 7-13
  REWRITE statement, 7-6
  summary, 1-6
  summary of access statements, 7-2
  UPDATE statement, 7-12

Beginning key of reference, 9-22, 9-24

BILD$R,
  arguments, 18-8, 18-9
  calling sequence, 18-8
  error messages, 18-14 to 18-16, B-8 to B-11
  introduction, 18-2, 18-8

BINARY, 3-3

BIND,
  CBL, 6-3
  F77, 5-9
  FTN, 5-8
  PL/I, 8-3
  VRPG, 9-2, 9-3

Block size,
  defining, 14-14
  specifications, 14-14

Buffer Management, 13-3

Building a MIDASPLUS file (See KBUILD)

C

C,
  introduction, G-1
  using an interlude routine, G-2

C_PRMO system, 16-2

Call interface with C, G-4
  introduction, G-1
  using an interlude routine, G-2

Call interface with Pascal,
  introduction, H-1
  using an interlude routine, H-2

Calling sequence, general (FORTRAN), 5-10

CBL,
  BIND, 6-3

CBL compiler, 6-1

Chained Files (VRPG),
  definition, 9-7
  errors, 9-12

Cleaning up a MIDASPLUS file (See MPLUSCLUP)

CLOSE statement,
  BASIC/VM, 7-4
  COBOL, 6-11
  RELATIVE file, 6-30

CLOSM$ routine,
  arguments, 5-14
  calling sequence, 5-14

COBOL interface,
  access modes, 6-18
  adding records, 6-21
  changing search indexes, 6-20
  declaring RELATIVE KEY, 6-28
  file type, 3-3
  introduction, 6-1
  introduction to direct access, 6-28
  keyed reads, 6-19
  partial key access, 6-20
  reading duplicates, 6-20
  sequential access, E-5
  sequential reads, 6-18
  summary, 1-6

COBOL Statements,
  AT END clause, 6-11
  CLOSE, 6-11
  DELETE, 6-22, 6-23
  INVALID KEY clause, 6-12
  OPEN, 6-9
  REWRITE, 6-22
  SELECT (RELATIVE file), 6-29
  START, 6-15
  USE AFTER, 6-13
  WRITE, 6-21

COBOL Status Codes, B-16, B-17

Command files, 16-1, 16-2

Communications array, 5-2

Compiling and loading,
  CBL, 6-3
  F77, 5-9
  FTN, 5-8
  PL/I, 8-3
  VRPG, 9-2, 9-3

Components of MIDASPLUS, 16-3

Concurrency,
  errors (VRPG), 9-12
  introduction, E-1
  locked records, E-1
  rules, E-7

Configuration,
  display, 13-6
  parameters, 16-5 to 16-8

Configuration parameters, 16-5 to 16-8

Continuation lines, 9-23, 9-24

COUNT (CREATK), 2-12, 14-2

Creating a file, 2-1
  (See also CREATK)
  from PL/I, 8-3

CREATK, 2-1
  ADD option, 2-12, 14-8
  and Variable-length record file, 2-5
  block size specifications, 14-14
  COUNT option, 2-12, 14-2, 14-3
  DATA option, 2-12, 14-5, 14-9
  defining block size, 14-14
  dialogs, 2-1, 2-2
  direct access dialog (minimum options), 2-9
  examine an existing template, 14-1
  explanation of options, 2-12
  EXTEND option, 2-12, 14-10
  extended options dialog, 14-14, 14-15
  FILE option, 2-12
  FILE READ/WRITE locks, 2-3
  FUNCTION prompt, 14-1
  GET option, 2-13
  HELP option, 2-13
  index block levels, 14-14
  INITIALIZE option, 2-13
  keyed access dialog (minimum options), 2-6
  MODIFY option, 2-13, 14-13
  optional features, 2-11
  PRINT option, 2-13, 14-2, 14-3
  QUIT option, 2-13
  sample file, 2-4
  SIZE option, 2-13, 14-5, 14-6
  USAGE option, 2-13, 14-6
  VERSION option, 2-14, 14-7

Current file position,
  VRPG, 9-8

Current record,
  FORTRAN, 5-11
  PL/I, 8-5
  VRPG, 9-8

Customer Service, D-1

D

DATA (CREATK), 2-11, 14-10

DATA DIVISION Requirements,
  CBL compiler, 6-7

Data file, 1-2

Declaring data size (PL/I), 8-7

DEFINE FILE statement, 7-3, 7-4

Defining an INDEXED MIDASPLUS
    file (COBOL), 6-4

DELET$ routine,
  arguments, 5-45, 5-46
  calling sequence, 5-46
  deleting duplicates, 5-46
  deleting secondary index
    entries, 5-46
  direct access, 5-46
  locating record to delete,
    5-46
  removing a record and all keys,
    5-46

DELETE statement,
  COBOL, 6-22, 6-23
  PL/I, 8-15
  RELATIVE file, 6-34

DELETED RECORDS, E-4

Deleting,
  MIDASPLUS file, (See also
    KIDDEL)
  records (VRPG), 9-10
  secondary index entries, 5-46

Demand Files (VRPG), 9-7

Dialog,
  CREATK, 2-6 to 2-11
  extended options, 2-1, 14-14
    to 14-16
  guidelines, 2-2
  KBUILD, 3-9 to 3-11
  KIDDEL, 11-2

Dialog (continued)
  MDUMP, 10-4, 10-5
  minimum options, 2-1
  MPACK, 15-4, 15-5

Direct access,
  array format, 5-3
  DELET$ routine, 5-46
  Dialog (Minimum Options), 2-9
  file structure, 4-3
  FORTRAN, 5-2
  LOCK$ routine, 5-40
  overview, 1-4, 4-2, 4-3, 5-23
  VRPG, 9-20

Directory protection, 16-3
  MIDASPLUS*, 16-4

Disk space, saving, 2-5

Duplicates,
  deleting (FORTRAN), 5-46
  retrieving (FORTRAN), 5-35

DYNAMIC Access Mode, 6-19

E

ERROPN,
  calling sequence, F-1
  introduction, F-1
  using the routine, F-2

Error handling,
  BASIC/VM, 7-4
  COBOL, 6-11
  offline build routines, 18-5
  VRPG, 9-11

Error log, system,
  configuration parameters, 16-7
  purpose, 16-11
  UFD, 16-12

Error messages,
  BILD$R, 18-14 to 18-16, B-8 to
    B-11
  KBUILD, 3-22 to 3-24, B-1 to
    B-3
  KIDDEL, 11-3, B-5
  KX$CRE, B-7
  MDUMP, 10-7, 10-8, B-5

Error messages (continued)
MPACK, 15-10, 15-11, B-6, B-7
PRIBLD, 18-14 to 18-16, B-8 to B-11
SECBLD, B-8 to B-11
SPY, B-5, B-6

Error reporting, KBUILD, 3-8

Event sequence flag, 18-3 to 18-5

Examining a file (CREATK), 14-2

Execute-only MIDASPLUS, 1-5
description, 1-5
installation, 16-1

EXTEND (CREATK), 2-11, 14-10

Extended options dialog, 2-1, 14-14 to 14-16

F

F77,
BIND, 5-9

FILE (CREATK), 2-11

File access methods, 1-4
direct, 1-4
keyed-index, 1-4

File addition specifications,
VRPG, 9-6

File Description,
format in CBL, 6-8

File designation restrictions,
VRPG, 9-6

File key, 2-3

FILE POSITION, (COBOL), 6-14

FILE READ/WRITE Locks, 2-3

FILE SECTION, 6-7

File Specification Statement, 9-22

File type specification (VRPG), 9-5

File unit management,
hard-coded files, I-1 to I-5
introduction, I-1
potential problems, I-6

FILE-CONTROL requirements, 6-5

FIND$ routine,
arguments, 5-26
array, 5-29
calling sequence, 5-25
direct access, 5-30
flags, 5-27
retrieval options, 5-28
specifying an index, 5-25

FL$BIT flag, 5-27, 5-33

FL$FST flag, 5-27, 5-33

FL$KEY flag, 5-21, 5-27, 5-29, 5-33, 5-43

FL$NXT flag, 5-27, 5-33

FL$PLW flag, 5-27, 5-33

FL$PRE flag, 5-33

FL$RET flag, 5-21, 5-27, 5-33

FL$SEC flag, 5-27, 5-33

FL$UKY flag, 5-27, 5-29, 5-33

FL$ULK flag, 5-43

FL$USE flag, 5-21, 5-27, 5-33, 5-43

Flags,
default setting, 5-8
for ADD1$, 5-21
for FIND$, 5-27
for NEXT$, 5-33
for UPDAT$ routine, 5-43
introduction, 5-4, 5-5
meanings, 5-6, 5-7

Flags (continued)
names, 5-6, 5-7
OFF, 5-8
ON, 5-8
settings, 5-6, 5-7

Fork system calls, PRIMIX, 4-3

FORTRAN interface, 1-5, 5-1
adding data record, 5-21
adding secondary index entries, 5-22
BIND, 5-8, 5-9
communications array, 5-2
current record, 5-1
direct access, 5-2
direct access array format, 5-3
flags, 5-4 to 5-7
general calling sequence, 5-10
$INSERT Mnemonics, 5-4
introduction, 5-1
optional arguments, 5-11
record locking, 5-10
redundant primary keys, 5-21
subroutines, 5-9, 5-10
summary, 1-5

FORTRAN routines,
ADD1$, 5-17, 5-19 to 5-22
CLOSM$, 5-14
DELET$, 5-45, 5-46
FIND$, 5-25 to 5-30
GDATA$, 5-35 to 5-37
LOCK$, 5-37 to 5-40
NEXT$, 5-31 to 5-35
NTFYM$, 5-15, 5-16
OPENM$, 5-12, 5-13
UPDAT$, 5-41 to 5-43

FTN,
BIND, 5-8

FTNBIN, 3-3

Function Call, 13-5

FUNCTION prompt (CREATK), 14-1

G

GDATA$ routine,
arguments, 5-36
calling sequence, 5-35

General calling sequence (FORTRAN), 5-10

GET (CREATK), 2-13

H

Hard-coded files, E-6, E-7, I-1 to I-5

HELP (CREATK), 2-13

I

IBM System/34 functionality, 9-30

IMIDASPLUS, 16-3

Imperative-statement, 6-11

Index block levels, 14-14, 14-15

INDEXED files (COBOL), 6-1

INITIALIZE (CREATK), 2-13

Initializing MIDASPLUS, 16-4

Input files,
introduction, 3-2
location of keys, 3-4
multiple files, 3-4
record compatibility, 3-4
rules, 3-3
sort requirements, 3-6
sorted, 3-5

$INSERT Mnemonics, 5-4

Installing MIDASPLUS, 16-1

Interlude routine, G-2

Internal Error Codes, B-14

INVALID KEY clause, 6-12

K

KBUILD,
  adding secondary index entries, 3-7
  alternatives to, 3-22
  building direct access files, 3-19
  dialog, 3-9 to 3-11
  error messages, 3-22 to 3-24, B-1 to B-3
  error reporting, 3-8
  input files, 3-2, 3-3
  introduction, 3-1
  milestone reporting, 3-8, 3-9
  supported input file types, 3-3
  variable-length record files, 3-6

Key errors (PL/I), 8-18

Key fields, 1-2

Key option, 8-10

Key types, 2-2

Key values,
  specifying, 5-28

Keyed Access Dialog (Minimum Options), 2-6

Keyed reads,
  COBOL, 6-19
  PL/I, 8-11

Keyed-index, 5-3
  access, 1-4
  adds (FORTRAN), 5-19
  array format, 5-3

KEYFROM option, 8-7

Keys, 1-2

KEYTO option, 8-10

KIDALB, 16-3

KIDDEL,
  dialog, 11-2
  error messages, 11-3, B-5
  introduction, 11-1

KX$CRE,
  arguments, 17-2
  calling sequence, 17-1
  error codes, 17-5, 17-6, B-7
  flags arguments, 17-2, 17-3
  introduction, 17-1

KX$RFC,
  arguments, 17-7, 17-8
  calling sequence, 17-7
  introduction, 17-6
  pridef and secdef flags, 17-8
  user-supplied arguments, 17-7

KX$TIM,
  calling sequence, F-2
  introduction, F-2
  using the routine, F-3

L

Language Access, 4-2

Language Dependencies (COBOL), 6-2

Language group summary, 1-5, 1-6

Layout of a BANK file, 2-4

Loading records (VRPG), 9-10

Locating the record to delete, 5-46

Location of keys, 3-4

LOCK$ routine,
  arguments, 5-38, 5-39
  array values, 5-39, 5-40
  calling sequence, 5-37
  direct access, 5-40
  specifying a key, 5-39

Locked records, E-1
  PL/I, 8-21

M

MDUMP,
  dialog, 10-4, 10-5
  error messages, 10-7, 10-8, B-5
  introduction, 10-1
  options, 10-1, 10-2
  sequential dump file, 10-3
  status and descriptive messages, 10-5 to 10-7, B-3, B-4

MIDASERR (BASIC/VM), 7-5

MIDASPLUS* UFD, 16-1

MIDASPLUS.INITINSTALL.COMI, 16-1

MIDASPLUS.SHARE.COMI, 16-2

MIDASPLUSEX.INITINSTALL.COMI, 16-1

Milestone reporting, 3-8, 3-9, 10-7

Minimum options dialog, 2-1

Miscellaneous Error Codes, B-12, B-14, B-15

MODIFY (CREATK), 2-13, 14-13

Monitoring a MIDASPLUS file (See SPY)

MPACK,
  abnormal termination, 15-5
  dialog, 15-4, 15-5
  error messages, 15-10 to 15-12, B-6, B-7
  functions and options, 15-1, 15-2
  introduction, 15-1
  milestone reports, 15-2
  MPACK mode, 15-3
  UNLOCK, 15-2

MPACK mode,
  ALL option, 15-3
  DATA option, 15-3
  index-number option, 15-3

MPLUS.CONFIG, 16-3, 16-4

MPLUSCLUP,
  introduction, 12-1
  options, 12-2
  remote cleanup, 12-2

MSGCTL, 16-9

Multiple input files, 3-4

N

Networking MIDASPLUS, 16-10

NEXT$ routine,
  arguments, 5-32
  array settings, 5-34
  buffer size, 5-33
  calling sequence, 5-31
  flags, 5-33

NPX slave, 16-10

NTFYM$ routine,
  arguments, 5-15, 5-16
  calling sequence, 5-15

O

Offline build routines,
  BILD$R, 18-2, 18-8, 18-9
  error handling, 18-6
  error messages, 18-14 to 18-16
  event sequence flag, 18-3, 18-4, 18-6
  guidelines, 18-2
  introduction, 18-1
  PRIBLD, 18-2, 18-5, 18-6
  restrictions, 18-3
  SECBLD, 18-2, 18-6, 18-7

Offline create routines,
  introduction, 17-1
  KX$CRE, 17-1 to 17-6
  KX$RFC, 17-6 to 17-8

ON ERROR statement, 7-4

ONKEY function, 8-19

Open Modes, 6-9

OPEN statement,
  COBOL, INDEXED file, 6-30
  COBOL, RELATIVE file, 6-9

Opening and closing MIDASPLUS
    files, 5-11

OPENM$ routine,
  arguments, 5-13
  calling sequence, 5-13
  keys, 5-12

Optional CREATK Features, 2-11

P

Partial key access,
  COBOL, 6-20
  FIND$, 5-28

Pascal,
  introduction, H-1
  using an interlude routine,
    H-2

PL/I Interface,
  access mode statements, 8-5
  accessing CREATK, 8-17
  BIND, 8-3
  combining DCL and OPEN, 8-4
  conversion, 8-2
  creating a MIDASPLUS file, 8-3
  current record, 8-5
  declaring data size, 8-7
  DELETE and the current record,
    8-6
  DELETE statement, 8-15
  error trapping, 8-19
  File I/O concepts, 8-5
  initial current record, 8-6
  introduction, 8-1

PL/I Interface (continued)
  keyed reads, 8-11
  KEYFROM option, 8-7
  language limitations, 8-2
  locked errors, 8-21
  locked records, 8-6
  OPEN statement, 8-4
  record errors, 8-21
  REWRITE KEY option, 8-14
  REWRITE statement, 8-13
  sequential reads, 8-11
  summary, 1-6
  WRITE statement, 8-6, 8-7

Positioning the file,
  VRPG, 9-8

PRIBLD,
  arguments, 18-6
  calling sequence, 18-5
  error messages, 18-14 to
    18-16, B-8 to B-11
  introduction, 18-2, 18-5

pridef, 17-3 to 17-5

PRIMIX, 4-3

PRIMOS error messages, C-1 to
    C-6

PRIMOS.COMI file, 16-2

PRINT (CREATK), 2-13, 14-2 to
    14-4

Process Waits, 13-5

Programming Error Codes, B-13

Q

QUIT (CREATK), 2-13

R

RAF, 9-7

RANDOM file,
  keyed reads, 6-33
  reading current record, 6-33

Read errors (VRPG), 9-12

READ statement,
  COBOL (INDEXED) file, 6-17 to 6-20
  COBOL (RELATIVE) file, 6-32
  PL/I, 8-10

READ/WRITE Error Codes, B-12, B-13

Reading a File,
  COBOL, 6-17 to 6-20
  duplicates (COBOL), 6-20
  FORTRAN, 5-24
  key values (PL/I), 8-12
  PL/I, 8-9
  records, VRPG, 9-8

Record compatibility, 3-4

Record errors (PL/I), 8-21

RECORD KEY, 6-4

Record locking,
  COBOL, 6-14
  FORTRAN, 5-10

Record Locks, 13-6
  display, 13-2

Records,
  adding (VRPG), 9-9
  adding to a MIDASPLUS file, 14-10
  deleting, 9-10
  loading, 9-10
  updating, 9-10

Redundant primary keys, 5-21

RELATIVE file,
  accessing, 6-30
  adding records, 6-31
  DELETE statement, 6-34
  opening and closing, 6-30
  READ statement, 6-32
  REWRITE statement, 6-33, 6-34

RELATIVE file (continued)
  sequential read, 6-32
  WRITE statement, 6-31, 6-32

RELATIVE KEY, declaring, 6-28

Remote cleanup, 12-2

REMOVE statement,
  BASIC/VM, 7-13

REPORTING PROBLEMS, D-1

Retrieval Options (FIND$), 5-28

Return code values, 5-24

Rewrite KEY option, 8-14

REWRITE statement,
  BASIC/VM, 7-6
  COBOL, 6-22
  INDEXED file, 6-33
  PL/I, 8-13
  RELATIVE file, 6-34

RPG, 3-3

Runtime error codes,
  internal, B-14
  miscellaneous, B-12, B-14, B-15
  programming, B-13
  READ/WRITE, B-12, B-13

S

Sample MIDASPLUS file, 1-3, 2-4

SECBLD,
  arguments, 18-7
  calling sequence, 18-7
  error messages, 18-14 to 18-16, B-8 to B-11
  introduction, 18-2, 18-7

secdef, 17-3 to 17-5

Secondary data, 2-7, 2-10

Secondary index entries,
  deleting, 5-46

SELECT statement,
  ALTERNATE RECORD KEY clause, 6-7
  defining in a RELATIVE file, 6-29
  FILE STATUS clause, 6-7
  format for INDEXED file, 6-5
  RECORD KEY clause, 6-6

SEQFLG (See Event sequence flag)

SEQUENTIAL ACCESS, E-5

Sequential Access Mode, 6-18

Sequential Dump File (MDUMP), 10-3

Sequential reads,
  COBOL, 6-18
  PL/I, 8-11

Sequential record retrieval, 5-34

SETLL, 9-7

SHARE.COMI, 16-2

Sharing MIDASPLUS, 16-2

SIZE (CREATK), 2-13, 14-5, 14-6

Snooze, 13-5, 13-6

Sorted input files, 3-5

Specification Statements,
  summary, 9-1

SPY,
  Awaken, 13-5
  configuration display, 13-6
  error messages, 13-13, B-5, B-6
  function call, 13-5
  introduction, 13-1
  Keys of locked records display, 13-12
  main menu, 13-2
  per-user configuration, 13-13
  process waits, 13-5
  product information, 13-3
  record locks, 13-2, 13-6

SPY (continued)
  Snooze, 13-5, 13-6
  statistics display, 13-3
  subfile to fileunit translation, 13-4
  system configuration, 13-7
  user interface, 13-1

START statement, 6-15, 6-16

Statement fields,
  VRPG, 9-5

Static on-unit, 12-1

STATISTICS DISPLAY, 13-3

Status and descriptive messages,
  MDUMP, 10-5 to 10-7, B-3, B-4

Status codes, B-16, B-17

Storing primary keys in record (PL/I), 8-9

Subfile to Fileunit Translation, 13-4

Subroutines, 5-10

Summary of COBOL statements, 6-4

SYSCOM>KEYS.INS.FIN, 5-4

SYSCOM>KEYS.INS.PL1, 5-4

SYSCOM>PARM.K.INS.FIN, 5-4

SYSCOM>PARM.K.INS.PL1, 5-4

System Configuration, 13-7

System error log, 16-11

T

Template,
  examine, 14-1
  function of, 1-4
  introduction, 1-4
  modifying, 14-7

TEXT, 3-3

U

Unit Utilization, I-1

UNLOCK (MPACK), 15-2

UPDAT$ routine,
  arguments, 5-42
  array, 5-41
  calling sequence, 5-41
  flags, 5-43
  unlock only, 5-41

Updating records,
  FORTRAN, 5-37
  VRPG, 9-10

USAGE (CREATK), 2-13, 14-6

USE AFTER Statement, 6-13

V

Variable-length record files, 3-6
  and ADD1$, 5-17
  and BILD$R, 18-9
  and PRIBLD, 18-5
  building, 3-6
  changing size limits, 2-5, 14-11
  checking size limits, 2-5, 14-4
  creating, 2-5
  maintaining, 2-5
  setting size limits, 2-5, 14-12

VERSION (CREATK), 2-14, 14-7

VKDALB, 16-3

VLRs (See Variable-length record files)

VRPG interface,
  accessing primary or secondary files, 9-7
  adding records, 9-9
  alternate file processing, 9-30 to 9-32
  beginning key of reference, 9-22, 9-24
  BIND, 9-2, 9-3
  chain errors, 9-12
  chained files, 9-7
  concurrency errors, 9-12
  continuation lines, 9-23, 9-24
  current file position, 9-8
  current record, 9-8
  DELETE, 9-10, 9-19
  demand files, 9-7
  direct access, 9-20
  error handling, 9-11
  file addition specifications, 9-6
  file descriptive specifications, 9-3
  file designation restrictions, 9-6
  file type specification, 9-5
  IBM System/34 functionality, 9-30
  introduction, 9-1
  language-dependent features, 9-2
  load, 9-10, 9-14
  multiple key processing, 9-20, 9-22 to 9-30
  positioning the file, 9-8
  RAF, 9-7
  random reads, 9-9
  read errors, 9-12
  reading records, 9-8
  record locked error, 9-11
  sequential reads, 9-9
  SETLL, 9-7
  Specification Statement summary, 9-1
  statement fields, 9-5
  summary, 1-6
  updating records, 9-10

W

Word, 2-2, 2-6

WRITE statement,
COBOL, 6-21
PL/I, 8-7

SURVEYS

READER RESPONSE FORM

MIDASPLUS User's Guide
DOC9244-2LA

Your feedback will help us continue to improve the quality, accuracy, and organization of our publications.

1. How do you rate this document for overall usefulness?

   ☐ *excellent* ☐ *very good* ☐ *good* ☐ *fair* ☐ *poor*

2. What features of this manual did you find most useful?

   _____
   _____
   _____
   _____
   _____
   _____

3. What faults or errors in this manual gave you problems?

   _____
   _____
   _____
   _____
   _____
   _____

4. How does this manual compare to equivalent manuals produced by other computer companies?

   ☐ *Much better* ☐ *Slightly better* ☐ *About the same*
   ☐ *Much worse* ☐ *Slightly worse* ☐ *Can't judge*

5. Which other companies' manuals have you read?

   _____
   _____

Name:_____ Position:_____

Company:_____

Address:_____

_____

_____ Postal Code:_____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

First Class Permit #531 Natick, Massachusetts 01760

BUSINESS REPLY MAIL

Postage will be paid by:

Prime™

Attention: Technical Publications
Bldg 21
Prime Park, Natick, Ma. 01760

READER RESPONSE FORM

MIDASPLUS User's Guide
DOC9244-2LA

Your feedback will help us continue to improve the quality, accuracy, and organization of our publications.

1. How do you rate this document for overall usefulness?

   ☐ *excellent* ☐ *very good* ☐ *good* ☐ *fair* ☐ *poor*

2. What features of this manual did you find most useful?

   _____
   _____
   _____
   _____
   _____
   _____

3. What faults or errors in this manual gave you problems?

   _____
   _____
   _____
   _____
   _____
   _____

4. How does this manual compare to equivalent manuals produced by other computer companies?

   ☐ *Much better* ☐ *Slightly better* ☐ *About the same*
   ☐ *Much worse* ☐ *Slightly worse* ☐ *Can't judge*

5. Which other companies' manuals have you read?

   _____
   _____

Name:_____ Position:_____

Company:_____

Address:_____

_____

_____ Postal Code:_____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

First Class Permit #531 Natick, Massachusetts 01760

BUSINESS REPLY MAIL

Postage will be paid by:

Prime™

Attention: Technical Publications
Bldg 21
Prime Park, Natick, Ma. 01760

READER RESPONSE FORM

MIDASPLUS User's Guide
DOC9244-2LA

Your feedback will help us continue to improve the quality, accuracy, and organization of our publications.

1. How do you rate this document for overall usefulness?

   ☐ *excellent* ☐ *very good* ☐ *good* ☐ *fair* ☐ *poor*

2. What features of this manual did you find most useful?

3. What faults or errors in this manual gave you problems?

4. How does this manual compare to equivalent manuals produced by other computer companies?

   ☐ *Much better* ☐ *Slightly better* ☐ *About the same*
   ☐ *Much worse* ☐ *Slightly worse* ☐ *Can't judge*

5. Which other companies' manuals have you read?

Name:_____ Position:_____

Company:_____

Address:_____

_____

_____ Postal Code:_____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

First Class Permit #531 Natick, Massachusetts 01760

BUSINESS REPLY MAIL

Postage will be paid by:

Prime™

Attention: Technical Publications
Bldg 21
Prime Park, Natick, Ma. 01760

DOC9244-2LA