

C - F I L E

COMPAS Pascal File Access Manager

Version 1.0

R E F E R E N C E M A N U A L

Copyright (C) 1983,1984
Poly-Data microcenter ApS
Aaboulevarden 13
DK-1960 Copenhagen V

C - F I L E

COMPAS Pascal File Access Manager

Version 1.0

R E F E R E N C E M A N U A L

Copyright (C) 1984

Poly-Data microcenter ApS
Aaboulevarden 13
DK-1960 Copenhagen V

COPYRIGHT

Copyright (C) 1983, 1984 by Poly-Data microcenter ApS. All rights reserved. No part of this publication may be copied, duplicated or otherwise distributed, in any form or by any means, without the prior written permission of Poly-Data microcenter ApS, Aaboulevarden 13, DK-1960 Copenhagen V, Denmark.

DISCLAIMER

Poly-Data microcenter ApS makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Further, Poly-Data microcenter ApS reserves the right to revise this publication without obligation of Poly-Data microcenter ApS to notify any person of such revision.

TRADEMARKS

COMPAS, COMPAS Pascal, COMPAS-80 and COMPAS-86 are trademarks of Poly-Data microcenter ApS. CP/M, CP/M-80 and CP/M-86 are trademarks of Digital Research Inc. MS-DOS is a trademark of Microsoft Inc.

Poly-Data microcenter ApS
Aaboulevarden 13
DK-1960 Copenhagen V, DENMARK

Telephone: +1 35 61 66
Telex: 16600 FOTEX DK, Att: microcenter

TABLE OF CONTENTS

0 Introduction 2

1 Overview of C-FILE 3

1.1 Main components 3
1.2 Function summary 3
1.2.1 Data file initialization and update 3
1.2.2 Index file initialization 4
1.2.3 Index file update 4
1.2.4 Index file search 4
1.3 C-FILE types and variables 4
1.4 General notes on C-FILE 5
1.5 Using C-FILE 6
1.6 Program structure with C-FILE 8

2 C-FILE routines 10

2.1 Data file support routines 10
2.1.1 The MAKEFILE procedure 11
2.1.2 The OPENFILE procedure 11
2.1.3 The CLOSEFILE procedure 11
2.1.4 The ADDREC procedure 12
2.1.5 The DELETEREC procedure 12
2.1.6 The GETREC procedure 12
2.1.7 The PUTREC procedure 13
2.1.8 The FILELEN function 13
2.1.9 The USEDRECS function 13
2.2 Index file support routines 13
2.2.1 The INITINDEX procedure 15
2.2.2 The MAKEINDEX procedure 15
2.2.3 The OPENINDEX procedure 16
2.2.4 The CLOSEINDEX procedure 16
2.2.5 The ADDKEY procedure 16
2.2.6 The DELETEKEY procedure 17
2.2.7 The FINDKEY procedure 17
2.2.8 The SEARCHKEY procedure 18
2.2.9 The NEXTKEY procedure 18
2.2.10 The PREVKEY procedure 19
2.2.11 The CLEARKEY procedure 19

3 Programming examples 20

4 C-FILE error handling 24

Section 0

Introduction

C-FILE is a complete set of utility routines which allow you to create and maintain data files and index files from your COMPAS Pascal programs. The routines require from 4K to 9K of code space, excluding buffer areas, and they are entirely written in COMPAS Pascal.

C-FILE data files contain fixed length data records, exactly the same way as a COMPAS Pascal data file (FILE OF type). Data file records must be at least eight bytes long, but there is no upper limit to their length. C-FILE automatically reuses previously deleted data file records before expanding a data file.

Index files contain key values (strings) and their associated data record numbers. C-FILE organizes index files using a height balanced multiway tree structure (also known as a B-Tree structure) to guarantee a minimum number of disk accesses to search an index file. Assume that an index file contains 50,000 key values, organized with up to 32 keys per node. Then, in a worst case situation, C-FILE will need only 3 disk accesses to locate a specific key!

Index file records (nodes) are buffered using a last-recently-used priority scheme to furthermore reduce disk accesses (whenever an index record is already held within a C-FILE node buffer, a disk access is eliminated). The node buffers are shared between all index files, so only one buffer area need be reserved.

For a complete description of the B-Tree approach, we recommend the following books:

Wirth, Niklaus: Algorithms + Data Structures = Programs. Prentice-Hall, 1976.

Knuth, D.E.: The Art Of Computer Programming, Volume 3: Sorting and Searching. Addison-Wesley Publishing, 1973.

Section 1

Overview of C-FILE

1.1 Main components

C-FILE is supplied as four COMPAS Pascal source files. The C-FILE modules are included in the compilation of an application program through the {$I filename} directive of the COMPAS compiler. The C-FILE files are:

DATMAN.PAS — Contains the basic data/index file setup and maintenance routines. This module must always be included before any other C-FILE modules.

NPFKEY.PAS — Contains the NEXTKEY, PREVKEY, FINDKEY and SEARCHKEY routines. Together these routines constitute the search functions of C-FILE.

ADDKEY.PAS — Contains the ADDKEY routine. The ADDKEY routine is used to add keys to index files.

DELKEY.PAS — Contains the DELETEKEY routine. The DELETEKEY routine is used to delete keys from index files.

The DATMAN module must always be included in a program that uses C-FILE and it must always be the first module. NPFKEY, ADDKEY and DELKEY may however be included or omitted as required, and their ordering is of no importance.

1.2 Function summary

C-FILE routines fall into four categories:

- Data file initialization and update routines.
- Index file initialization routines.
- Index file update routines.
- Index file search routines.

The functions of each of these categories are summarized in the following sections. The descriptions also specify which C-FILE module must be included in your program in order to access a specific function.

1.2.1 Data file initialization and update

MAKEFILE — Creates a data file (DATMAN).

OPENFILE — Opens an existing data file (DATMAN).

CLOSEFILE — Closes a data file (DATMAN).

ADDREC — Allocates a new data record. ADDREC automatically reuses previously deleted data records before extending the data file (DATMAN).

DELETEREC Deletes a data record. The record is entered in the list of deleted records, and will be reused by ADDREC before the data file is extended (DATMAN).

GETREC Reads a data record (DATMAN).

PUTREC Writes a data record (DATMAN).

FILELEN Returns the total number of records in a data file, including deleted and reserved records (DATMAN).

USEDRECS Returns the number of data records in use in a data file (DATMAN).

1.2.2 Index file initialization

INITINDEX Initializes the C-FILE index file buffers (DATMAN).

MAKEINDEX Creates an index file (DATMAN).

OPENINDEX Opens an existing index file (DATMAN).

CLOSEINDEX Closes an index file (DATMAN).

1.2.3 Index file update

ADDKEY Adds a new key string and its associated data record number to an index file (ADDKEY).

DELETEKEY Deletes a key string and its associated data record number from an index file (DELKEY).

1.2.4 Index file search

FINDKEY Finds an index file entry which is equal to a given search string (NPFKEY).

SEARCHKEY Finds the first index file entry which is equal to or greater than a given search string (NPFKEY).

NEXTKEY Returns the index entry which immediately follows the last entry located by a search routine (NPFKEY).

PREVKEY Returns the index entry which immediately precedes the last entry located by a search routine (NPFKEY).

CLEARKEY Sets the index pointer to point at the beginning or end of an index file for sequential processing (DATMAN).

1.3 C-FILE types and variables

In addition to the basic data/index file maintenance routines, the C-FILE DATMAN module also defines some global types and variables. They are:

DATAFILE — This type identifier is used to declare data file variables. All C-FILE data files are declared using this identifier, even though their data records are not of the same type and size.

INDEXFILE — This type identifier is used to declare index file variables.

OK — OK is a boolean variable which is used to return the status of some C-FILE routines.

1.4 General notes on C-FILE

C-FILE data files may contain up to 65536 records and the record size may, in theory, be up to 64K bytes. In contrast to ordinary COMPAS Pascal data files, C-FILE data file variables are always declared using the DATAFILE type (the actual record size is specified at run-time when a new data file is created). C-FILE data files always have fixed size records. The minimum data record length is 8 bytes. The first record of a data file (record number 0) is reserved for system use (the ADDREC routine will automatically reserve this record).

C-FILE creates index files using a height-balanced multiway tree structure known as a B-Tree. This structure guarantees a minimum number of disk accesses to locate a specific key, and furthermore eliminates the need to reorganize index files. Whenever a key is added or deleted, C-FILE automatically adjusts the index file.

The index files built and maintained by C-FILE contain key values (strings) and data record numbers which specify the location in the data file of the information identified by the associated key. With only a key value, an index file makes it possible for an application program to locate the associated data record.

C-FILE makes no assumptions about the relationship between index files and data files. It is up to your application program to extract key information from a data record and enter it into an index along with a data record number. As a consequence, several index files may reference one data file.

It is recommended that you include the key values both in the data file records and the index files, since this will largely ease the task of reconstructing a corrupted index. It also enables you to use computed index keys, i.e. keys which are computed from one or more fields in a data record. An example: Supposing that your application program is to maintain a mailing list - The keys stored in the index might then be the last name of each entry in the data file, converted to upper case. If a search string is also converted to upper case, C-FILE will find the appropriate entry regardless of whether it was entered using upper and/or lower case letters, but the data record will still contain the data exactly as it was entered. In general, key values should only be omitted from the data file records if you have a very limited amount of disk storage.

1.5 Using C-FILE

As stated earlier the C-FILE routines are made available to your application program by including one or more of the four C-FILE modules. The DATMAN module should always be included before other C-FILE modules since it contains the basic disk file interface. Following DATMAN you may then include the NPFKEY, ADDKEY and DELKEY modules as required.

Prior to including DATMAN, some integer constants must be declared. They are:

MAXRSIZE — Maximum record length. MAXRSIZE should be set to the size (in bytes) of the largest data records your program is to process (i.e. if your program is to process two data files with record sizes of 72 and 140 bytes, then MAXRSIZE should be set to 140).

MAXKSIZE — Maximum key length (an integer between 1 and 255). MAXKSIZE should be set to the largest maximum key length of the index files your program is to process (i.e. if your program is to process three index files with maximum key lengths of 16, 10, and 25, then MAXKEYLEN should be set to 25).

NODESIZE — Node size. The maximum number of key entries allowed in each index file record (node). The node size must be the same for all index files to be processed by your program. NODESIZE should be an even number between 4 and 254. For further details, see below.

NODEHALF — Half node size. Half the size given by NODESIZE.

NBUFSIZE — Node buffer size. The number of index file records (nodes) that can be kept within memory at one time. The minimum value is 3. In general, increasing NBUFSIZE will speed up the system, since the probability that an index file record is already within memory when it is to be processed gets larger.

MAXDEPTH — The maximum depth of the index file structure. This constant applies to all index files to be processed by your program. For a calculation of MAXDEPTH see below.

The maximum number of index nodes, K, to be searched to find a specific key in an index file with E keys is:

```
K = LOG(E) / LOG(NODESIZE * 0.5)
```

where NODESIZE is the node size. Thus, for large nodes a search requires less disk accesses, and thus less time, than for small nodes (the time required to perform a search within the node, once it has been read into memory, is of no significance compared to the time it takes to read the node from the disk). The MAXDEPTH parameter required by DATMAN corresponds to the integer part of K plus one, so once you have established the node size and the maximum number of elements you will allow, you may compute MAXDEPTH from:

```
MAXDEPTH = INT(LOG(E) / LOG(NODESIZE * 0.5)) + 1
```

Note that larger values of MAXDEPTH will only require very little extra memory. It is therefore recommended that you add 2 or 3 instead of 1, to be on the safe side.

The number of bytes, N, occupied by each node in an index file is:

```
N = (KSIZE + 5) * NODESIZE + 3
```

where KSIZE is the maximum key length for that index file. The maximum number of bytes, D, occupied by an index file is:

```
D = N * E / (NODESIZE * 0.5)
```

This formula for D is based on a worst case assumption. Experience shows that the multiplication factor for NODESIZE normally lies around 0.75.

The number of bytes, M, occupied in memory by the C-FILE node buffer is:

```
M = ((MAXKSIZE + 5) * NODESIZE) + 3) * NBUFSIZE
```

where MAXKSIZE is the largest key length used by the index files to be processed, and NBUFSIZE is the maximum number of nodes that may be kept within memory at the same time (note that this must be at least three).

It is difficult to devise a general method for calculating the optimal values of NODESIZE and NBUFSIZE.

NODESIZE usually lies between 16 and 32, depending on the maximum key size and the number of keys in the index. Smaller values will result in poor performance as to the time required to look up a key, and larger values will require too much memory for node buffers.

The minimum resonable value for NBUFSIZE is that of MAXDEPTH. If NBUFSIZE is less than MAXDEPTH, the same node will have to be read several times to traverse the B-Tree sequentially. In general, NBUFSIZE should be as large as possible. Specifically, if NBUFSIZE is larger than NODESIZE, C-FILE may keep the root node and the entire first level of the B+ tree within memory, thus reducing the number of disk reads required to look up a key by at least one.

A skeleton program which uses C-FILE might look like this:

```
PROGRAM yourprog;

CONST
  maxrsize = 132;     {Maximum record size}
  maxksize = 25;      {Maximum key length}
  nodesize = 24;      {Node size}
  nodehalf = 12;      {Half node size}
  nbufsize = 8;       {Node buffer size}
  maxdepth = 5;       {Maximum tree depth}
```

```
{$I DATMAN}          {Must go before other C-FILE modules}

{Include other C-FILE modules}

{Your declarations}

BEGIN

  {Your main program}

END.
```

1.6 Program structure with C-FILE

In most cases an application program will incorporate one or more of the following four functions: Add data records, retrieve data records, update data records, and delete data records. Also, prior to processing any data, an application program must prepare (open) the necessary index and data files, and, before terminating, the program must close these files.

The initialization stage consists of calling MAKEFILE or OPENFILE for each data file to be used, and MAKEINDEX or OPENINDEX for each index file. In addition, INITINDEX must be called to initialize the index file manager routines (this need only be done once, for instance at the very beginning of the application program, and only if any of the index routines are to be used).

To add a data record, first input the record, and add it to the data file using ADDREC. Next, compute a key value for the new record (usually, this will simply be one of the fields in the data record), and call ADDKEY to add it to the index file. On calling ADDKEY, you must pass the data record number returned by ADDREC, so that it may be entered into the index along with the key. If your program is maintaining more than one index, call ADDKEY for each index file, passing the same data record number on every call.

To locate a key value, you may use the FINDKEY, SEARCHKEY, NEXTKEY, PREVKEY, and CLEARKEY routines to search the index (or if you have several indexes, to search one of them), and once the key has been found you may use GETREC to read the associated data record from the data file.

When a data record is to be deleted, first determine its key (for instance using FINDKEY, SEARCHKEY, NEXTKEY, PREVKEY, and CLEARKEY as described above), and then call DELETEKEY to delete it from the index file. DELETEKEY will return the data record number contained in the key which was deleted. If you are maintaining more than one index, then read the data record at that position (using GETREC), and use it to compute and delete the keys in the other index files. Finally, call DELETEREC to delete the data record from the data file.

On updating a data record, the changes made to the record may at times affect the key value(s). If so, you should call DELETEKEY to delete the old key and ADDKEY to add the new key. If you have more than one index file, this procedure must be repeated for each file. Finally you should call PUTREC to write the updated record to the data file.

Prior to terminating your program must call CLOSEFILE for each data file in use, and CLOSEINDEX for each index file.

Section 2

C-FILE routines

This section describes the routines made available to the COMPAS Pascal programmer by the C-FILE utility.

Most C-FILE routines return a status value using a boolean variable called OK, which is automatically declared by C-FILE. For instance, the OPENFILE routine sets OK to TRUE if the file to be opened was found and FALSE if not. In case of severe and/or unrecoverable errors, a procedure called IOCHECK, which is located in the DATMAN module, gains control. Normally IOCHECK outputs the name of the file, the record number and the error code, and then terminates the program. You may however write your own IOCHECK procedure. For more details on this, please refer to section 4.

2.1 Data file support routines

C-FILE provides data file support routines for the following tasks:

- Create, open, and close data files.
- Add and delete records to and from data files.
- Read and write records from and to data files.
- Report the size of data files.

All routines described in section 2.1 are contained in the DATMAN module.

In contrast to ordinary COMPAS Pascal data files, C-FILE data file variables are always declared using the DATAFILE type (the actual record size is specified at run-time when a new data file is created). C-FILE data files always have fixed size records. The minimum data record length is 8 bytes. There is no upper limit to the data record size. The first record of a data file (record number 0) is reserved for system use (the ADDREC routine will automatically reserve this record).

As mentioned earlier C-FILE will automatically reuse previously deleted data records before expanding a data file when new records are added. This is achieved by maintaining a linked list of deleted data records: When a data record is deleted, its first two bytes form a pointer to the next deleted record (-1 indicates that the record is the last record in the list). Since a zero pointer (two bytes of zero) never occurs, you may, if you wish, reserve the first two bytes of each data record, and set them to zero when you add a record to the file. This will enable you to distinguish used records from deleted records if you process the file without reference to an index (such processing will be necessary to reconstruct a corrupted index file).

2.1.1 The MAKEFILE routine

```
PROCEDURE MAKEFILE(VAR DATF: DATAFILE; FNAM: STR14;
                   RECLEN: INTEGER);
```

MAKEFILE creates a new data file and prepares it for processing by the C-FILE routines. The parameters passed to MAKEFILE are:

DATF — The data file variable to be prepared for access. DATF must be of type DATAFILE.

FNAM — A string expression of up to 14 characters specifying the name of the new disk file.

RECLEN — The record length in bytes. The minimum record length is 8.

On exit the OK flag is set to TRUE if the file was created successfully. If OK is FALSE there was no room on the disk to create a new file.

2.1.2 The OPENFILE routine

```
PROCEDURE OPENFILE(VAR DATF: DATAFILE; FNAM: STR14;
                   RECLEN: INTEGER);
```

OPENFILE opens an existing data file and prepares it for processing by the C-FILE routines. The parameters passed to OPENFILE are:

DATF — The data file variable to be prepared to access. DATF must be of type DATAFILE.

FNAM — A string expression of up to 14 characters specifying the name of an existing disk file.

RECLEN — The record length in bytes. The record length must be the same as the one used when the file was created.

On exit the OK flag is set to TRUE if the file was found and opened successfully. Otherwise OK is set to FALSE.

2.1.3 The CLOSEFILE routine

```
PROCEDURE CLOSEFILE(VAR DATF: DATAFILE);
```

CLOSEFILE closes the data file given by DATF. If you make any changes to a data file, always call CLOSEFILE for that file before terminating your program. Failing to do so may cause data to be lost, not to mention possible corruption of the data file structure.

2.1.4 The ADDREC routine

```
PROCEDURE ADDREC(VAR DATF: DATAFILE; VAR DRN: INTEGER;
                 VAR BUFFER);
```

ADDREC adds a new record to a C-FILE data file. If any previously deleted records are available, they are automatically reused before the disk file is expanded. The parameters passed to ADDREC are:

DATF — The data file to which the record is added.

DRN — Data record number. ADDREC returns the data record number of the newly allocated data record using this variable parameter. You should pass this data record number to the ADDKEY index routine when you add a key value for the data record.

BUFFER — A variable containing the data record to be added. Since BUFFER is an untyped parameter, ADDREC will accept any variable in its place. It is up to you to make sure that the variable passed is of the proper type.

Note that ADDREC does not return a status value. It only returns if the data record was added to the file successfully. If an I/O error occurs, IOCHECK will gain control and terminate the program. We suggest that you call USEDRECS before calling ADDREC to make sure that there is enough room on the disk.

2.1.5 The DELETEREC routine

```
PROCEDURE DELETEREC(VAR DATF: DATAFILE; DRN: INTEGER);
```

DELETEREC deletes a data record from a C-FILE data file. The record is entered into the deleted data record list, so that it may be reused by ADDREC before the data file is expanded. The parameters passed to DELETEREC are:

DATF — The data file from which the record is deleted.

DRN — The data record number. If you are maintaining an index, this number should be passed to the DELKEY index routine when the key value is deleted.

2.1.6 The GETREC routine

```
PROCEDURE GETREC(VAR DATF: DATAFILE; DRN: INTEGER;
                 VAR BUFFER);
```

GETREC reads a specified data record into memory. The following parameters are passed to the routine:

DATF — The data file from which the record is read.

DRN — The data record number.

BUFFER A variable to read the data into. Since BUFFER is an untyped parameter, GETREC will accept any variable in its place. It is up to you to make sure that the variable passed is of the proper type.

2.1.7 The PUTREC routine

```
PROCEDURE PUTREC(VAR DATF: DATAFILE; DRN: INTEGER;
                 VAR BUFFER);
```

PUTREC writes a data record to a specific position in a data file. The following parameters are passed to the routine:

DATF The data file to which the record is written.

DRN The data record number.

BUFFER The variable from which the data is written. Since BUFFER is an untyped parameter, GETREC will accept any variable in its place. It is up to you to make sure that the variable passed is of the proper type.

2.1.8 The FILELEN function

```
FUNCTION FILELEN(VAR DATF: DATAFILE): INTEGER;
```

FILELEN returns the number of data records allocated to the data file given by DATF. The length returned by FILELEN includes the reserved record at the beginning of the file (record 0) as well as all deleted records.

2.1.9 The USEDRECS function

```
FUNCTION USEDRECS(VAR DATF: DATAFILE): INTEGER;
```

USEDRECS returns the number of data records in DATF that contain valid data. In contrast to FILELEN, this function does not include reserved and deleted records.

2.2 Index file support routines

C-FILE provides index file support routines for the following tasks:

- Create, open and close index files.
- Add and delete keys to and from index files.
- Search for keys in index files.
- Read through index files forwards and backwards.

Index files are declared using the INDEXFILE type identifier. Entries in an index file consist of a key string and a data record number. Key strings may assume values of type string. The maximum length of the key strings is determined when the index file is opened. If a key string is too long, it will be truncated when it is passed on to an index file support routine.

Duplicate keys

There are many applications, such as an index based on last names, where key strings are not guaranteed to be unique. In other words, duplicate keys may occur. C-FILE only allows duplicate keys if the STATUS parameter in a call to MAKEINDEX or OPENINDEX is 1.

When C-FILE adds records to index files with duplicate keys, equal keys are ordered by their data record numbers, so that key entries with lower record numbers appear first. Normally this will correspond to the order in which the keys are entered, since new data records are usually added to the end of data files.

The search routines (FINDKEY and SEARCHKEY) will always locate the first key entry, i.e. the key entry with the lowest data record number.

To delete a key from an index file with duplicate keys, it is not sufficient simply to specify the key string, since this string may identify several entries. To select a specific entry you must also specify the data record number. The DELETEKEY routine will delete the key entry only if the key string and the data record number match the values found in the index file.

Numeric keys

If your application program requires numeric key values, you must convert these numeric values to strings before passing them on to C-FILE. There are basically two methods of doing this.

The simplest approach is to convert the numeric value to its ASCII string representation using the STR standard procedure. If you use this method, the resulting strings must be right justified (this is easily accomplished by specifying a field width in the call to STR). The main disadvantage to this method is that the key length must be set to the maximum number of digits that may occur as opposed to the number of bytes required to store the number in its binary format.

The second approach only applies to integers, and it takes advantage of the compactness of integers in binary format. The routines shown below may be used to "pack" and "unpack" integers into strings. ITOS converts an integer to a string, and STOI vice versa. The strings returned by ITOS are two characters long, and the strings passed to STOI must likewise be two characters long.

```
FUNCTION ITOS(N: INTEGER): STR2;
BEGIN
  N:=N+$8000; ITOS:=CHR(HI(N))+CHR(LO(N));
END;

FUNCTION STOI(S: STR2): INTEGER;
BEGIN
  N:=SWAP(ORD(S[1]))+ORD(S[2])+$8000;
END;
```

The above routines operate on signed integers (-32768..32767). If the integers are to be interpreted as unsigned quantities, simply remove the additions of $8000.

Data file splitting

A C-FILE index file must be contained in a single disk file. Data files may however be spread over more than one disk file, the only limitation being that the total number of records may not exceed 65536. The splitting of data files is quite simple to implement and best illustrated by an example: Assume that each data file can hold 10000 records and that we need to store up to 30000 records. Thus, we will require three data files. When a record is added to the first file, the data record number is entered directly into the index file. However, when records are added to the second and the third file, we add 10000 or 20000 to the record number. Later when the index file is read, record numbers may be divided by 10000 to determine in which files the records reside, and the remainders of the divisions are the actual data record numbers. Note that since the first record number in a data file is 1, you must subtract 1 before dividing and add 1 to the remainder to produce correct results.

2.2.1 The INITINDEX procedure

```
PROCEDURE INITINDEX;
```

INITINDEX initalizes the tables used by the index file routines. It must be called before other index file routines. Only one call is required, and it is usually placed at the very beginning of the application program that uses C-FILE.

2.2.2 The MAKEINDEX procedure

```
PROCEDURE MAKEINDEX(VAR IDXF: INDEXFILE; FNAM: STR14;
                    KEYLEN,STATUS: INTEGER);
```

MAKEINDEX creates a new index file and prepares it for processing by the C-FILE index file routines. The new index file will initially be empty. The following parameters are passed to the procedure:

IDXF — The index file variable to be prepared for access. It must be of type INDEXFILE.

FNAM — A string expression of up to 14 characters specifying the disk file name.

KEYLEN — The maximum length of the key strings to be stored in this file.

STATUS — File status. 0 indicates that duplicate keys are not allowed. 1 means that duplicate keys may occur.

On exit the OK flag is set to TRUE if the file was created successfully. If OK is FALSE there was no room on the disk to create a new file.

2.2.3 The OPENINDEX procedure

```
PROCEDURE OPENINDEX(VAR IDXF: INDEXFILE; FNAM: STR14;
                    KEYLEN,STATUS: INTEGER);
```

OPENINDEX opens an existing index file and prepares it for processing by the C-FILE index file routines. The following parameters are passed to the procedure:

IDXF — The index file variable to be prepared for access. It must be of type INDEXFILE.

FNAM — A string expression of up to 14 characters specifying the disk file name.

KEYLEN — The maximum length of the key strings to be stored in this file. KEYLEN must be the same as the one used when when the index file was created.

STATUS — File status. 0 indicates that duplicate keys are not allowed. 1 means that duplicate keys may occur.

On exit the OK flag is set to TRUE if the file was found and opened successfully. Otherwise OK is set to FALSE.

2.2.4 The CLOSEINDEX procedure

```
PROCEDURE CLOSEINDEX(VAR IDXF: INDEXFILE);
```

CLOSEINDEX closes a C-FILE index file. If you make any changes to an index file, always call CLOSEINDEX for that file before terminating your program. Failing to do so may cause data to be lost, not to mention possible corruption of the index file structure.

2.2.5 The ADDKEY procedure

```
PROCEDURE ADDKEY(VAR IDXF: INDEXFILE;
     VAR DRN: INTEGER; VAR KEY);
```

ADDKEY adds a key string to an index file. To use ADDKEY you must include the ADDKEY module. The following parameters are passed to the procedure:

IDXF — The index file to which the key string is to be added.

DRN — The data record number to be associated with the key. Usually DRN is a data file record number returned by ADDREC.

KEY — The key string. Since KEY is an untyped parameter you may pass string variables of any string type to ADDKEY. It is however up to you to ensure that the parameter is a string variable - other variables and string expressions are not allowed. If the KEY parameter is longer than the maximum key length for IDXF it will be truncated to the maximum length.

On exit the OK flag is set to TRUE if the key string was added successfully. OK returns FALSE if you try to add a duplicate key when such keys are not allowed, i.e. when the STATUS parameter in the call to OPENINDEX or MAKEINDEX was 1.

2.2.6 The DELETEKEY procedure

```
PROCEDURE DELETEKEY(VAR IDXF: INDEXFILE;
      VAR DRN: INTEGER; VAR KEY);
```

DELETEKEY removes a key string from an index file. To use DELETEKEY you must include the DELKEY module. The following parameters are passed to the procedure:

IDXF — The index file from which the key string is to be removed.

DRN — The data record number associated with the key to be deleted. If duplicate keys are not allowed in the index file, DRN need not be initialized. If duplicate keys are allowed, DELETEKEY however needs the data record number to tell the keys from each other. To determine the data record number, you may for example use SEARCHKEY in connection with NEXTKEY and PREVKEY. DRN always returns the data record number of the key that was deleted.

KEY — The key string to be deleted. The KEY parameter must be a string variable. If it is longer than the maximum key length for IDXF it will be truncated to the maximum length.

On exit OK is set to TRUE if the key was removed successfully. OK returns FALSE if the key was not found. If duplicate keys are allowed, OK will return FALSE if a matching data record number was not found even though the key string existed.

2.2.7 The FINDKEY procedure

```
PROCEDURE FINDKEY(VAR IDXF: INDEXFILE;
      VAR DRN: INTEGER; VAR KEY);
```

FINDKEY returns the data record number associated with a specific key string in an index file. To use FINDKEY you must include the NPFKEY module. FINDKEY locates the entry in the index file that exactly matches the string passed as the KEY parameter. If the index file contains duplicate keys, FINDKEY will always locate the first key. The following parameters are passed to the procedure:

IDXF — The index file to search.

DRN — If the key is found, its associated data record number is returned in this parameter.

KEY The key string to search for. The KEY parameter must be a string variable. If it is longer than the maximum key length for IDXF it will be truncated to the maximum length.

On exit OK is set to TRUE if a matching key was found. Otherwise OK is set to FALSE.

2.2.8 The SEARCHKEY procedure

```
PROCEDURE SEARCHKEY(VAR IDXF: INDEXFILE;
      VAR DRN: INTEGER; VAR KEY);
```

SEARCHKEY returns the data record number associated with the first entry in an index file that is equal to or greater than a specific key value. To use SEARCHKEY you must include the NPFKEY module. SEARCHKEY can be used to locate an entry in an index file when only the first part of the key value is known. If the index file contains duplicate keys, SEARCHKEY will always locate the first key. The following parameters are passed to the procedure:

IDXF The index file to search.

DRN If the key is found, its associated data record number is returned in this parameter.

KEY The key string to search for. The KEY parameter must be a string variable. If it is longer than the maximum key length for IDXF it will be truncated to the maximum length.

OK is always set to TRUE on exit, unless no keys are greater than or equal to the search key. In that case OK is set to FALSE.

2.2.9 The NEXTKEY procedure

```
PROCEDURE NEXTKEY(VAR IDXF: INDEXFILE;
      VAR DRN: INTEGER; VAR KEY);
```

NEXTKEY returns the data record number associated with the next entry in an index file. NEXTKEY also returns the key value in the KEY parameter. To use NEXTKEY you must include the NPFKEY module. The following parameters are passed to NEXTKEY:

IDXF An index file that has been prepared for sequential processing by a call to FINDKEY, SEARCHKEY or CLEARKEY.

DRN Returns the data record number associated with the key string.

KEY Returns the key string read from the next index entry.

On exit OK is set to TRUE unless no next index entry exists. In that case OK is set to FALSE. When OK returns FALSE, i.e. when the pointer is at the end of the index, NEXTKEY will return the first entry in the index if it is called again.

Before the very first call to NEXTKEY for a given index file or after the index file is updated using ADDKEY or DELETEKEY, one of the other index search functions, except PREVKEY, must be called. The search functions establish the internal pointer used by NEXTKEY and PREVKEY for sequential processing.

2.2.10 The PREVKEY procedure

```
PROCEDURE PREVKEY(VAR IDXF: INDEXFILE;
     VAR DRN: INTEGER; VAR KEY);
```

PREVKEY returns the data record number associated with the preceding entry in an index file. PREVKEY also returns the key value in the KEY parameter. To use PREVKEY you must include the NPFKEY module. The following parameters are passed to PREVKEY:

IDXF — An index file that has been prepared for sequential processing by a call to FINDKEY, SEARCHKEY or CLEARKEY.

DRN — Returns the data record number associated with the key string.

KEY — Returns the key string read from the preceding index entry.

On exit OK is set to TRUE unless no preceding index entry exists. In that case OK is set to FALSE. When OK returns FALSE, i.e. when the pointer is at the beginning of the index, PREVKEY will return the last entry in the index if it is called again.

Before the very first call to PREVKEY for a given index file or after the index file is updated using ADDKEY or DELETEKEY, one of the other index search functions, except NEXTKEY, must be called. The search functions establish the internal pointer used by NEXTKEY and PREVKEY for sequential processing.

2.2.11 The CLEARKEY procedure

```
PROCEDURE CLEARKEY(VAR IDXF: INDEXFILE);
```

CLEARKEY sets the index file pointer to the beginning or end of an index file. Following a call to CLEARKEY, a call to NEXTKEY will return the first entry in the index file, and a call to PREVKEY will return the last entry in the index file.

When a C-FILE index file is processed sequentially it may be compared to a ring. When the index file pointer is at the end of the file, a request to read the next entry will return the first entry in the file. Likewise, when the index file pointer is at the beginning of the file, a request to read the preceding entry will return the last entry in the file. In fact, the beginning and the end are at the same point in the file.

Section 3

Programming examples

The examples shown in this section demonstrate some common operations on a C-FILE data base. In this case the data base is very simple. The records in the data file contain only two fields, and there is only one index file. However, the principles are the same for more complex data bases with larger records and several index files.

The following record type is used for records in the data file:

```
TYPE
  PHONEREC = RECORD
               STATUS: INTEGER;
               PHONE: STRING[15];
               NAME: STRING[30];
             END;
```

The PHONE field will be used as the key in the index. Note the inclusion of a STATUS field in the beginning of the record. If this field is set to zero when a record is added to the data file, it will enable you to distinguish used records from deleted records when processing the data file without reference to the index file (for instance to rebuild a corrupted index).

The C-FILE constants are defined as follows:

```
CONST
  MAXRSIZE = 49;      (* Max record size *)
  MAXKSIZE = 15;      (* Max key size *)
  NODESIZE = 16;      (* Node size *)
  NODEHALF = 8;       (* Half node size *)
  NBUFSIZE = 5;       (* Node buffer size *)
  MAXDEPTH = 5;       (* Max index depth *)
```

MAXRSIZE is the size of the PHONEREC type, and MAXKSIZE is the maximum length of the string contained in the PHONE field.

The variables used in the examples are defined as follows:

```
VAR
  DATF: DATAFILE;
  IDXF: INDEXFILE;
  PREC: PHONEREC;
  PN: STRING[15];
  CH: CHAR;
  D: INTEGER;
```

The example shown below will create a data file called PHONE.DAT and an index file called PHONE.IDX. Note that the SIZE function may be used to specify the size of a data record.

```
MAKEFILE(DATF,'PHONE.DAT',SIZE(PREC));
IF OK THEN MAKEINDEX(IDXF,'PHONE.IDX',15,0);
IF OK THEN
BEGIN
  CLOSEFILE(DATF); CLOSEINDEX(IDXF);
END ELSE
WRITELN('Cannot create data files');
```

The following example is used to add records to the data base. Since duplicate phone numbers are not allowed, the STATUS parameter in the call to OPENINDEX is 0. A record is only added if a call to FINDKEY returns FALSE in OK, indicating that the key is not in the index. If ADDKEY were to be called right away, the program would have to call DELETEREC in case of a duplicate key. Otherwise, the data file would contain a record which was not in the index. Note that the STATUS field is set to zero before a new record is added so that active records can be told from deleted records when the data file is processed without reference to the index file.

```
INITINDEX;
OPENFILE(DATF,'PHONE.DAT',SIZE(PREC));
IF OK THEN OPENINDEX(IDXF,'PHONE.IDX',15,0);
IF OK THEN
BEGIN
  REPEAT
    WRITE('Phone number...: '); READLN(PN);
    PREC.PHONE:=PN; FINDKEY(IDXF,D,PN);
    IF OK THEN
    BEGIN
      WRITELN; WRITELN('Already assigned');
    END ELSE
    BEGIN
      WRITE('Name.....: '); READLN(PREC.NAME);
      PREC.STATUS:=0;
      ADDREC(DATF,D,PREC); ADDKEY(IDXF,D,PREC.PHONE);
    END;
    WRITELN;
    WRITE('Add another (Y/N)? '); READLN(CH);
    WRITELN;
  UNTIL NOT(CH IN ['Y','y']);
  CLOSEFILE(DATF); CLOSEFILE(IDXF);
END ELSE
WRITELN('Cannot open data files');
```

This example shows how to look up records via the index. FINDKEY is used to search the index, and if the key exists FINDKEY returns the data record number. The data record number is then used in a call to GETREC to read the actual data record.

```
INITINDEX;
OPENFILE(DATF,'PHONE.DAT',SIZE(PREC));
IF OK THEN OPENINDEX(IDXF,'PHONE.IDX',15,0);
IF OK THEN
BEGIN
  REPEAT
    WRITE('Phone number...: '); READLN(PN);
    FINDKEY(IDXF,D,PN);
    IF OK THEN
    BEGIN
```

```
        GETREC(DATF,D,PREC);
        WRITELN('Name.........: ',PREC.NAME);
      END ELSE
      BEGIN
        WRITELN; WRITELN('Key not found');
      END;
      WRITELN;
      WRITE('Scan again (Y/N)? '); READLN(CH);
      WRITELN;
    UNTIL NOT(CH IN ['Y','y']);
    CLOSEFILE(DATF); CLOSEFILE(IDXF);
  END ELSE
  WRITELN('Cannot open data files');
```

The following example is used to delete records. DELETEKEY is used to delete the specified key string. If the key is deleted successfully DELETEKEY returns the data record number associated with the key. This number is then used in a call to DELETEREC to delete the actual data record.

```
  INITINDEX;
  OPENFILE(DATF,'PHONE.DAT',SIZE(PREC));
  IF OK THEN OPENINDEX(IDXF,'PHONE.IDX',15,0);
  IF OK THEN
  BEGIN
    REPEAT
      WRITE('Phone number...: '); BUFLEN:=15; READLN(PN);
      DELETEKEY(IDXF,D,PN);
      IF OK THEN DELETEREC(DATF,D) ELSE
      BEGIN
        WRITELN; WRITELN('Key not found');
      END;
      WRITELN;
      WRITE('Scan again (Y/N)? '); READLN(CH);
      WRITELN;
    UNTIL NOT(CH IN ['Y','y']);
    CLOSEFILE(DATF); CLOSEFILE(IDXF);
  END ELSE
  WRITELN('Cannot open data files');
```

The CLEARKEY and NEXTKEY routines may be used to process the data base sequentially as shown below. Note that NEXTKEY returns FALSE in OK when all keys in the index are processed.

```
  INITINDEX;
  OPENFILE(DATF,'PHONE.DAT',SIZE(PREC));
  IF OK THEN OPENINDEX(IDXF,'PHONE.IDX',15,0);
  IF OK THEN
  BEGIN
    CLEARKEY;
    REPEAT
      NEXTKEY(IDXF,D,PN);
      IF OK THEN
      BEGIN
        GETREC(DATF,D,PREC);
        WRITELN(PREC.PHONE,'':16-LEN(PREC.PHONE),PREC.NAME);
      END;
    UNTIL NOT OK;
    CLOSEFILE(DATF); CLOSEFILE(IDXF);
  END ELSE
```

```
    WRITELN('Cannot close data files');
```

A data file can also be processed without reference to an index file. However this requires that a status field (an integer) is included as the first field of each data record. When a record is added to the data file (using the ADDREC routine) the application program must manually set the status field to zero. When the data file is processed sequentially, the status field will still be zero for a particular record if that record is not deleted. Other (non-zero) values indicate that the record is deleted. Note that processing starts at record number 1, which is actually the second record in the data file. The first record, i.e. record number 0 is reserved by C-FILE for various status informations.

```
OPENFILE(DATF,'PHONE.DAT',SIZE(PREC));
IF OK THEN
BEGIN
  D:=1;
  WHILE D<FILELEN(DATF) DO
  BEGIN
    GETREC(DATF,D,PREC); D:=D+1;
    IF PREC.STATUS=0 THEN
    WRITELN(PREC.PHONE,'':16-LEN(PREC.PHONE),PREC.NAME);
  END;
  CLOSEFILE(DATF);
END ELSE
WRITELN('Cannot open data file');
```

The following example shows how to rebuild a corrupted index file. The program reads the data file sequentially and adds the phone number of each active record to a new index file. If a duplicate key is encountered, an error message is output and the data record is deleted.

```
INITINDEX;
OPENFILE(DATF,'PHONE.DAT',SIZE(PREC));
IF OK THEN
BEGIN
  MAKEFILE(IDXF,'PHONE.IDX',15,0);
  IF OK THEN
  BEGIN
    D:=1;
    WHILE D<FILELEN(DATF) DO
    BEGIN
      GETREC(DATF,D,PREC);
      IF PREC.STATUS=0 THEN
      BEGIN
        ADDKEY(IDXF,D,PREC.PHONE);
        IF NOT OK THEN
        BEGIN
          WRITELN('Duplicate key. Record ',D);
          DELETEREC(DATF,D);
        END;
      END;
      D:=D+1;
    END;
  END ELSE
  WRITELN('Cannot create index file');
END ELSE
WRITELN('Cannot open data file');
```

Section 4

C-FILE error handling

C-FILE routines generate two types of errors: Non-fatal errors and fatal errors. Fatal errors cause the program to terminate, whereas non-fatal errors are reported to the program.

Non-fatal error conditions are reported through a boolean variable called OK, which is automatically declared by the DATMAN module. For instance OPENFILE returns FALSE in OK if the file specified was not found and FINDKEY returns FALSE if the key string was not found.

If a fatal error occurs, a routine called IOCHECK is invoked. IOCHECK is located in the DATMAN module. It prints an error code, a file name and a record number, and then terminates the program. Below follows an example of an error printout:

```
C-FILE I/O ERROR 10
File A:DATABASE.DAT Record 103
Program terminated
```

A C-FILE fatal error is actually equivalent to a COMPAS I/O error. Possible error codes are therefore the same as the ones listed in Appendix F in the COMPAS Pascal Programming Manual (note that C-FILE outputs the error code in decimal whereas COMPAS outputs it in hex).

In general, fatal errors only occur due to corrupted data and/or index files. A fatal error will however also occur if you try to expand a data file or an index file when there is no room on the disk.

If you want your program to come to a less abrupt halt in case of a fatal error, you may write your own IOCHECK routine. Typically, this is done by changing IOCHECK in the DATMAN module into a FORWARD declared procedure, with the body of the customized IOCHECK following in the application program. Note that IOCHECK should not return to the caller - its only purpose is to print an error message and terminate the program. If IOCHECK returns, the state of the system is undefined, and C-FILE is likely to crash sooner or later.

If required, a customized IOCHECK procedure may close files through calls to CLOSEFILE and CLOSEINDEX before it terminates the program. However, calls to CLOSEFILE and CLOSEINDEX may again result in errors, and in such cases IOCHECK must return to the caller to avoid an indefinite loop. The example shown below achieves this checking through a boolean variable.

To create a customized IOCHECK routine, replace IOCHECK in the DATMAN module with the following FORWARD declaration:

```
PROCEDURE IOCHECK(VAR DATF: DATAFILE; R: INTEGER);
FORWARD;
```

In the application program place the following declarations after the compiler directives that include the C-FILE modules:

```
VAR
  ERROR: BOOLEAN;

PROCEDURE IOCHECK;
BEGIN
  IF (IOSTS<>0) AND NOT ERROR THEN
  BEGIN
    WRITELN;
    WRITELN('C-FILE I/O ERROR ',IOSTS);
    WRITELN('File ',DATF.CNAM,'.',DATF.CTYP,' Record ',R);
    ERROR:=TRUE;
    IF DATOPEN THEN CLOSEFILE(DATF);
    IF IDXOPEN THEN CLOSEINDEX(IDXF);
    WRITELN('Program terminated');
    HALT;
  END;
END;
```

The ERROR variable should be initialized to FALSE at the beginning of the application program. If IOCHECK detects an error, it sets ERROR to TRUE, and further errors (originating from embedded calls to CLOSEFILE and CLOSEINDEX) are ignored.

DATF and IDXF in this case are the files operated upon by the program, and they are assumed to be declared elsewhere. DATF is of type DATAFILE and IDXF is of type INDEXFILE. DATOPEN and IDXOPEN are boolean variables that reflect the status of the two files. When a specific variable is TRUE, its associated file is assumed to be open and thus need to be closed if IOCHECK detects an error.