RCSL No: 31-D600

Edition: June 1980

Author: Inge Borch
Edith Rosenberg

Title:

RC8000 Indexed Sequential Files (ISQ)

A/S REGNECENTRALEN
af 1979

Keywords:

RC8000, Backing Storage Package, Indexed Sequential File, ALGOL.

Abstract:

This manual describes a specific structure of an indexed sequential file stored in a backing storage document and a set of RC8000 ALGOL procedures for processing such a file.

(56 printed pages)

Printed by A/S Regnecentralen af 1979, Copenhagen

Users of this manual are cautioned that the specifications contained herein are subject to change by RC at any time without prior notice. RC is not responsible for typographical or arithmetic errors which may appear in this manual and shall not be responsible for any damages caused by reliance on any of the materials presented.

FOREWORD

Second edition: RCSL No 31-D 558.
The present paper is a revised edition of RCSL No 55-D 99 (November 1970, Jørn Jensen) and is updated with changes from the version released in 1979. The changes mainly have consequences for the definition of the file head and the bucket head and make versions prior to 1979 incompatible with current ones. Correction lines in the left margin indicate changes of importance to 'old' users. The pure extensions to the system are described in RCSL No 31-D 601: Extensions to the Indexed Sequential Files System, April 1979, Inge Borch.

Acknowledgement: The system was designed and implemented in its first version by Jørn Jensen in 1970-71. Few systems have claimed less maintenance.

Inge Borch
A/S Regnecentralen, March 1979

Third edition: RCSL No 31-D 600.
This edition has been retyped but is similar to second edition apart from typographical corrections. The only important correction is the expression for computing "segsperbuckettable" (page 9), which is marked by a double correction line.

Edith Rosenberg
A/S Regnecentralen af 1979, June 1980

TABLE OF CONTENTS PAGE

TABLE OF CONTENTS (continued) PAGE

TABLE OF CONTENTS (continued) PAGE

APPENDICES:

1. INTRODUCTION 1.

An indexed sequential file is basically a sequential file, stored on a random access medium, and augmented by one or more levels of index tables to facilitate random access to records specified by a key.

With two levels, buckets and blocks, the search for a record with a specific key proceeds as follows:

A search for the key in the bucket table, which is common for the whole file, will yield a part of the file, the bucket, in which to continue the search.

Each bucket is preceded by a block table and a search in this will yield a part of the bucket, the block, in which the record may be found.

The inherent characteristics for this type of files are:

1) Fast sequential processing of the whole file, comparable to a straightforward sequential file.

2) Fast direct access for inspecting and updating of records specified by their keys.

3) Fast deletion of records.

4) Slow insertion of new records in a file, especially when the file is pretty full.

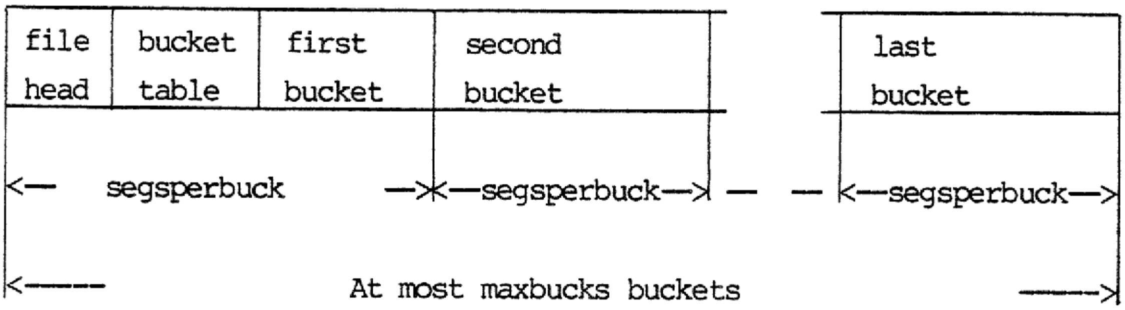
This paper describes the RC8000 ALGOL implementation of an indexed sequential file organization with two levels of index tables.

The system can be regarded as an extension of the set of the high level zone procedures and works within the same framework. It consists of a set of procedures to set up and process an indexed sequential file in an existing backing storage document which has been opened in a zone.

2. THE STRUCTURE OF AN INDEXED-SEQUENTIAL FILE ON THE BACKING STORAGE 2.

The file starts at segment zero of the area and consists of a file head, a bucket table, and a number of buckets. Each bucket except the first occupies segsperbuck consecutive segments.

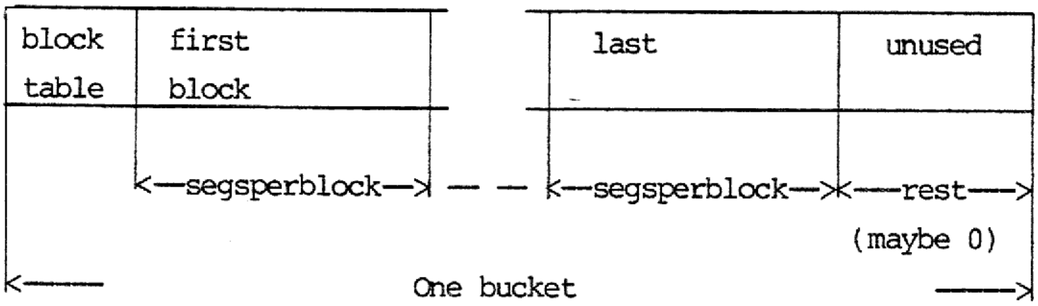
Picture of the file:



The file head and the bucket table occupy an integral number of segments each, and the first bucket occupies only what is left of the first segsperbuck segments.

Each bucket consists of a block table, which occupies an integral number of segments, followed by as many whole blocks as there is room for in the bucket, leaving a possible rest unused.
Each block occupies segsperblock consecutive segments.

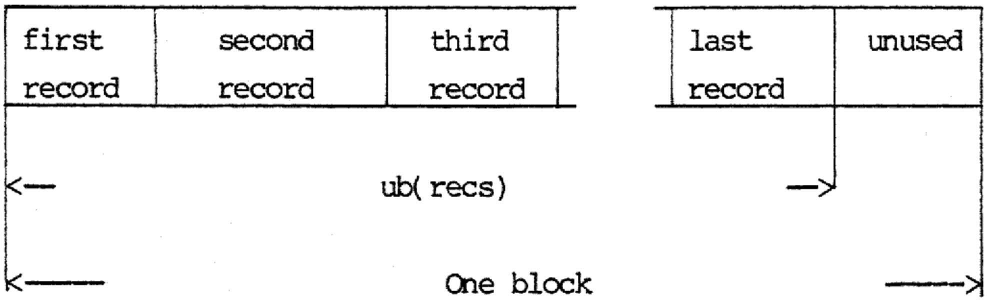
Picture of one bucket:



Each block consists of an integral number of records (possibly zero) stored tightly together in key order starting at the first byte of the block and leaving a possible rest unused.

ub(recs) denotes the number of halfwords used for records within a block (see section 2.1).

Picture of one block:



The file head describes the structure of record, blocks, and buckets in a form, which is convenient for the internal logic of the standard procedures processing the file.

The bucket table forms the first level of index tables and contains one entry for each bucket in the file describing the current contents of that bucket.

The block tables, one for each bucket, form the second level of index tables. The block table for a given bucket contains one entry for each block in the bucket describing the current contents of that block.

The structure and contents of records, index tables, and the file head are described below.

2.1 Records 2.1

Each record consists of zero or more user fields, a key consisting of an ordered set of key fields, and maybe a length field. The formats and contents of the user fields are irrelevant to the system. The key- and length-fields are described by code pieces in the file head. These descriptions are common for all records in the file.

2.1.1 Key Fields 2.1.1

The key is an ordered set of one or more key fields the value of which is unique identification of the record within the file. Each key field is characterized by a field type, which specifies the size of the key field and how the value of it is represented, and a relative position of the field within the record. The total number of key fields is denoted nkey.

The possible types, the number of halfwords in the corresponding key fields, and the values by which they are specified to the system (see section 6.5, head_file_i) are:

type:	number of halfwords:	value:
12-bit signed integer	1	± 1
integer	2	± 2
long	4	± 3
real	4	± 4

The sign of the type is used by the comparison rule, see below.

The relative position of a field is the byte number within the record of the last byte of the field, the first byte being byte one.

2.1.2 Comparison Rule 2.1.2

The keys of two records can be compared, i.e. the relations key(A) < key(B), key(A) = key(B), and key(A) > key(B) are defined for two records, A and B. If each key is composed of nkey keyfields then the comparison rule is defined by the following (not pure ALGOL) algorithm which compares the key fields, arithmetically according to type, two and two:

```
for i:= 1 step 1 until nkey do
   begin
   compare:= (keyfield(A,i)-keyfield(B,i))*sign(type(i));
   if compare <> 0 then i:= nkey
   end;
```

Compare now holds the result of the comparison and we define:

compare < 0 means key(A) < key(B).
compare = 0 means key(A) = key(B).
compare > 0 means key(A) > key(B).

stigende

Records are always stored in the file in ascending key order as defined by the above; i.e. in ascending order of the key field values for positive types, but in descending order of the key field values for negative types.

2.1.3 Length Field 2.1.3

The length field holds the record length, expressed as number of double word items, and is, just as a key field, characterized by a type and a relative position. Only non-negative types are meaningful for the length field.

If all records in the file have the same length, the length field may be absent. This is specified to the system by a type value = zero, in which case we have:

recordlength = maxreclength, see head_file_i (section 6.5.).

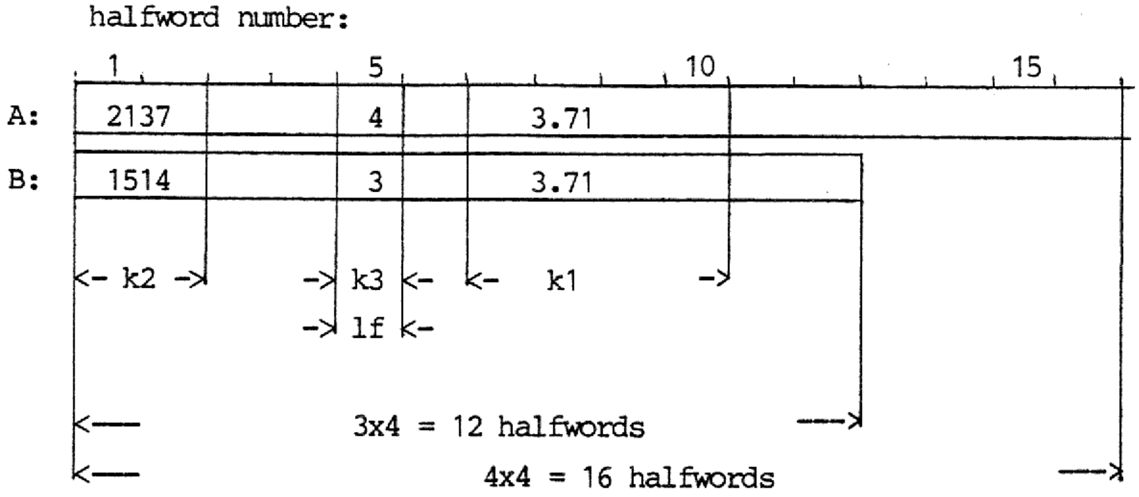
The different fields of a record may overlap each other in any manner as illustrated in the following example where the length field and the third key field occupy the same byte.

2.1.4 Example 2.1.4

Let the key- and length-fields be specified by

	type	relative position
1. key field	4	10
2. key field	-2	2
3. key field	-1	5
length field	1	5

then record A will precede record B in the following picture:



2.2 Block Tables 2.2

Each entry in a block table describes one block and consists of the following three fields:

ub(recs): An integer holding the number of halfwords occupied by records in the block.

sn(recs): An integer holding the segment number for the first segment of the block.
sn(recs) may thus be regarded as the identification of the physical block relative to the file-start.

kp(recs): A composite field consisting of the key fields of a record packed together in consecutive words and with a value such that:
kp(recs) > key(records preceding the block) and
kp(recs) <= key(first record in the block).
kp(recs) may thus be regarded as the identification of the logical block.

The size, in halfwords, of one entry in a block table, or the bucket table, see below, is given by:

entrysize = 4 + keypartsize, where:

keypartsize = 2 * number of words used for keyfields in a record.

In the above calculation of keypartsize two successive keyfields of type ± 1 are only counted as one word whereas a single keyfield of type ± 1 counts as a whole word. The algorithm is:

```
keypartsize:= 0;
for i:= 1 step 1 until nkey do
begin
  fieldsize:= abs type(i);  if fieldsize = 3 then fieldsize := 4;
  if fieldsize > 1 then keypartsize:= keypartsize + keypartsize mod 2;
  keypartsize:= keypartsize + fieldsize
end;
keypartsize:= keypartsize + keypartsize mod 2;
```

The block table for a non-empty bucket, i.e. a bucket which contains at least one record, consists of the entries describing non-empty blocks, stored in ascending kp-order, followed by the entries describing empty blocks. In these last entries only the value of sn is relevant as the contents of the block itself are undefined.

The size, in halfwords, of a block table is given by:

```
blocktablesize = entrysize * blocksperbuck, where

blocksperbuck  = segsize * segsperbuck//(segsize *
                 segsperblock + entrysize);

segsize = number of halfwords in one segment = 512.
```

A block table is stored in an integral number of segments:

```
segsperblocktable = (blocktablesize -1)//segsize + 1.
```

2.3 Bucket Table

Each entry in the bucket table describes one bucket and consists of the following three fields:

ub(blocks): An integer holding the relative byte address of the last non-empty entry in the block table for the bucket, the first entry having byte address zero; i.e.:
ub(blocks) = entrysize * (number of non-empty blocks - 1).

sn(blocks): An integer holding the segment number for the first segment of the blocktable for the bucket.
sn(blocks) may thus be regarded as the identification of the physical bucket relative to the filestart.

kp(blocks): A composite field consisting of the key field of a record packed together in consecutive words and with a value such that:
kp(blocks) > key(records preceding the bucket) and
kp(blocks) <= key(first record in the bucket).
kp(blocks) may thus be regarded as the identification of the logical bucket.

Entrysize and keypartsize are defined as for the block tables above.

The bucket table consists of a bucket table head followed by the entries describing non-empty buckets, stored in ascending kp-order, followed by the entries describing empty buckets. In these last entries only the value of sn is relevant as the contents of the bucket itself are undefined.

The bucket table head consists of five integer fields which describe the current contents of the bucket table and thereby of the whole file:

maxusedbucks: Number of relevant halfwords in the bucket table, including the bucket table head; i.e.:

```
maxusedbucks = entrysize * number of buckets which
                           are or have been non-empty during
                           the lifetime of the file + 30;
```

recbytes: Total number of halfwords occupied by records in the file.

noofrecs: Total number of records in the file.

ub(file): Relative address of the last non-empty entry in the bucket table, the first entry having halfword address zero; i.e.:

```
ub(file) = entrysize * (number of non-empty
                buckets -1).
```

sn(file): Segment number for the first segment of the bucket table. Note that maxusedbucks is the first word on this segment.

The size, in halfwords, of the bucket table is given by maxusedbucks, but it is stored in an integral number of segments which can hold a bucket table with maxbucks entries:

```
segsperbucktable = (entrysize * maxbucks + 30 - 1)//segsize + 1;
```

2.4 File Head 2.4

The file head describes the structure of the records, blocks, and buckets of the file as specified in the preceding sections. It is generated when the file is created (see chapter 4), and is unchanged on the backing storage during the lifetime of the file. It is read in to core and modified when the file is prepared for processing (see sections 5.2 and 5.3).
It holds the following five sections of information:

It holds the following five sections of information:

zonebufrefrel: An integer holding the relative address of the first halfword of fileparameters, see below, first halfword of zonebufrefrel being halfword one. It is used to facilitate references to fileparameters.

kp(save): A composite working field for holding the keypart of a record, size = keypartsize (see section 2.2).

savelength: A working field for holding the lengthfield of a record; zero, one, or two words depending on the type of the lengthfield.

recordcodes: The description of the key and lengthfields of a record in the form of code pieces for comparing and moving these fields. The formats and sizes depend on the specification of the key.

fileparameters: Parameters, working locations, and variables describing the records, blocks, and buckets in a format which is independent of the specific file and known by the procedures processing the file. When the file head is read into core some of these parameters are modified to absolute addresses which are used to reference other parts of the zonebuffer, the zone descriptor, and the share descriptors.

The details about the above sections are not given in this paper as they mainly are of interest for the understanding of the internal logic of the system.

The total size, in halfwords, of the filehead is the sum of the sizes of each of the above sections and has at present the value:

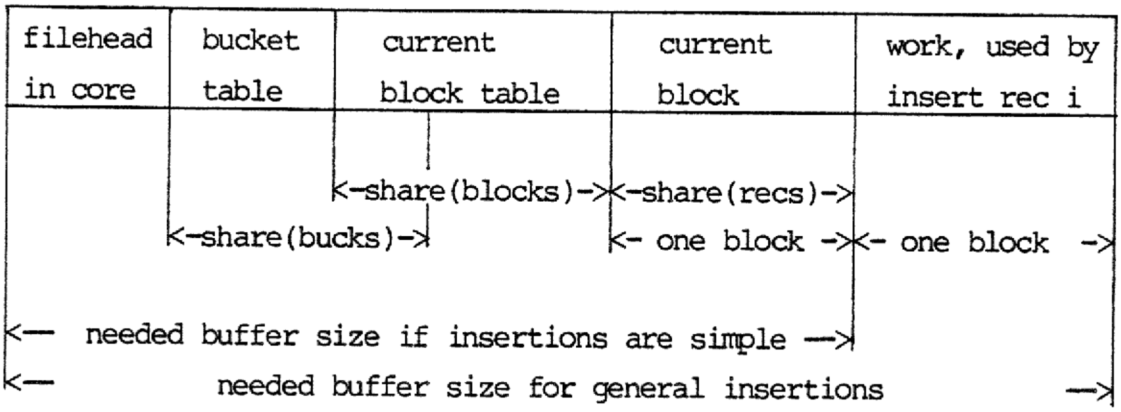
```
fileheadsize =
     2 +
     keypartsize +
     (if lengthtype = 0 then 0 else if lengthtype < 3 then 2 else 4) +
     nkey * 24 + number of type three keyfields * 8 +
     (keypartsize + 2)//4 * 4 + (if lengthtype = 0 then 6 else 14) +
     146;
```

The filehead is stored in an integral number of segments, starting at first word of the first segment of the area:

```
     segsperhead = (fileheadsize - 1)//segsize + 1.
```

3. AN INDEXED-SEQUENTIAL FILE IN THE ZONE BUFFER 3.

During the processing of a file, i.e. when a record is available (see sections 5.4 and Appendix A.2), the zone buffer holds in general the following five sections of information:



Filehead holds code pieces, absolute addresses, and other parameters used by the file_i procedures. It is read from the document and modified by init_file_i or start_file_i (see sections 5.2 and 5.3), and is never written back. It occupies only the necessary fileheadsize halfwords and normally not an integral number of segments as in the document.

Bucket table holds the bucket table from the document, including the bucket table head, but only with the number of buckets for which there is room in the document. The buckettablesize thus satisfies the condition:

maxusedbucks <= buckettablesize <= entrysize * maxbucks + 30

The bucket table is read by init_file_i or start_file_i and is only written back if the contents have been changed during the processing, i.e. if records have been deleted or inserted. The bucket table is described in the first share of the zone, denoted share(bucks), as segsperbucktable segments and may thus overlap the next share as shown.

Current block table holds the block table from the last accessed bucket. It occupies segsperblocktable segments and is described in the second share, denoted share(blocks). If the current blocktable has been changed, i.e. records have been inserted or deleted, it will be written back to the document before another block table is read in.

Current block holds the last accessed block from the last accessed bucket. It occupies segsperblock segments and is described in the third share, denoted share(recs). If the current block has been changed, i.e. records have been updated, inserted, or deleted, it will be written back to the document before another block is read in.

Work is an area which is only used by insert_rec_i when two blocks are needed in the core at the same time. The third share is then temporarily modified to describe this block. Work need not be present if only simple insertions of new records are needed (see section 6.8).

The total minimum size, in halfwords, of the zonebuffer is the sum of each of the above sections and has the value:

```
zonebuffersize =
   fileheadsize +
   entrysize * ((segsindocument - 1) // segsperbuck + 1) + 30 +
   segsize * segsperblocktable +
   segsize * segsperblock +
   (if simpleinsertions then 0 else segsize * segsperblock)
```

4. THE CREATION OF A FILE 4.

An empty indexed sequential file with a structure as described in chapter 1 is created by storing a filehead and a bucket table, describing an empty file, in the first segments of a backing storage area. The file can then later be initialized and processed as described in chapter 5.

4.1 The Area 4.1

The area must be a backing storage area with a segment length of 256 words. It must be opened and closed by explicit calls of the normal standard procedures, open and close, before and after use.

The size of the area is not used before the file is initialized. During creation the area needs therefore only be big enough to hold the file head and the bucket table head, see below.

4.2 The File Head 4.2

The file head will normally be generated directly into the area by a call of the external ALGOL procedure head_file_i, but it may also be copied from some other document, e.g. if more files with identical structure are needed.

4.3 Choice of Parameters to headfilei 4.3

The parameters of head_file_i (see section 6.5) determine the storage requirements and running characteristics of the file_i procedures and must be chosen with some care. The following is a survey of the influence of each of the parameters:

recdescr:

nkey: The number of keys determines the size of entries in the bucket table and the block tables and thus influences the size of share(bucks) and share (blocks), see below. The choice between fixed and variable recordlength has no significant influence on the running characteristics of the system.

maxreclength: Defines the maximum length (or fixed length) of a record, besides that it influences the strategy for elimination of overflow. If this parameter is chosen too large insert_rec_i will be forced to take a too pessimistic view on the amount of pushing together necessary, and the time used for non-simple insertions will be larger than necessary. In determining whether overflow occurs or not the actual record length is used and maxreclength has no influence. If a small part of a file consists of very long records it may be advantageous to split these to permit the system to run with a smaller value of maxreclength.

maxbucks: Is used to determine the size of the bucket table on the document. In core the size of the bucket table is determined by the size of the document. The search strategy in the bucket table is optimal when the documents contain maxbucks buckets and too large a value of maxbucks may cause a very slight decrease in the search efficiency.

segsperbuck:
segsperblock: These parameters (in connection with recdescr) determine the number of blocks per bucket and thus influence the size of the blocktables. Note that share(blocks) occupies an integral number of segments and that certain combinations of blocks per bucket and entrysize therefore give an inefficient utilization of core store. segsperblock defines the size of share(recs) and the work area. The overall search strategy will be optimal when the actual number of buckets and the number of blocks both are equal to maxbucks, but the effect on the search efficiency is negligible in almost all cases.
Segsperblock must be able to hold at least 2 records of maxreclength.

5. THE PROCESSING OF A FILE 5.

The system for processing a file with a structure as described in chapters 2 and 3 consists of one standard integer variable, result_i, and a number of standard procedures, in the following denoted the file_i procs.

The processing of the file may be split up in four phases:

opening,
initialization or start,
record processing, and
closing.

This chapter describes these four phases and the general rules for the use of the file_i procs.

5.1 Opening 5.1

The file is opened, i.e. connected with a zone, by a call of the normal RC8000 ALGOL standard procedure, open.

The minimum length of the zone buffer is a function of the structure of the file, as defined by the procedure head_file_i, the number of segments in the document, and whether or not the full facilities for the insertions of new records are needed. The exact length is given in chapter 3, but to avoid that the programs all should need to know the detailed structure of the file, the system has been augmented by an integer procedure, buf_length_i, which yields the needed length.

The number of shares in the zone must be three.

5.1.1 Example 5.1.1

The zone declaration and the open call for the file <:pip:> may look as follows:

```
begin
...
zone z(buf_length_i (<:pip:>, true), 3, stderror);
...
open(z, 4, <:pip:>, giveup);
...
```

5.2 Initialization 5.2

When a new file has been created it must be initialized with an initial set of records which have been sorted in ascending key order. When many records have been inserted by insert_i (see section 6.8), further insertions become impossible or their cost excessive indicating that the file should be reorganized. This is done by dumping all the records in the file in ascending key order and using this set of records to initialize the file.

This initialization is prepared by an open call, as described above, followed by a call of init_file_i which will:

read, check, and modify the file head,
set up an empty bucket table with as many buckets as there
    is room for in the document,
set the share descriptors of the zone to describe the three
    shares share(bucks), share(blocks), and share(recs)
    (see chapter 3).

The initialization itself is affected by successive calls of init_rec_i, each call adding one record to the file, and it must be terminated by a call of one of the procedures set_read_i, set_update_i, or set_put_i. The file is now ready for record processing with the first record of the file available as the zone record (see section 5.7).

5.2.1 The Initial Set of Records 5.2.1

The file should be initialized by as many records as possible because it is much more time consuming to insert unsorted records one at a time in an already initialized file.

If only a small set of records is available for initialization, they should reflect the final distribution of keys and they should be spread out uniformly through the file. This may be achieved through proper use of two of the parameters to init_file_i, the buckfactor and the blockfactor (see section 6.6).

buckfactor specifies the average number of blocks, useblocks, which init_rec_i should use in each bucket, where:

```
useblocks = buckfactor * blocksperbuck.
```

blockfactor specifies the average number of halfwords, usebytes, which init_rec_i should use for records in each block, where:

```
usebytes = segsperblock * segsize * blockfactor.
```

5.2.2 Example 5.2.2

The open call in example 4.1.1 may be followed by the call:

```
init_file_i (z, .5, .5)
```

which will specify that init_rec_i should only use half of the blocks in each bucket and half of the room in each used block. Thus only a quarter of the full capacity of the file can be used during initialization, but the unused capacity will be spread out through the file and thus facilitate later insertions of new records.

5.3 Start 5.3

When the file is non-empty, i.e. already has been initialized, processed, and closed, it is reopened for processing by an open call followed by a call of start_file_i which will:

read, check, and modify the file head,
read the bucket table, compare it with the number of segments in the document, protest if there are fewer buckets than last time the file was processed, and extend the bucket table if there are more,
set the share descriptors,
read the first block table and block, and
return with the first record of the file available as the zone record.

The file is now ready for record processing in read_only_mode, see below.

5.4 Record Processing 5.4

When the file has been properly initialized or started, the individual records can be handled by means of the following procedures:

get_rec_i: Makes a record with a specified key available.

next_rec_i: Makes the next record available.

delete_rec_i: Deletes the available record from the file and makes the next available.

insert_rec_i: Inserts a new record in its proper place in the file and makes it available.

This processing will take place in one of three modes:

read_only_mode: Records cannot be changed, blocks will only be read and not written.

update_mode: Records can be changed, all blocks which are read will also be written before a new block is read.

put_mode: Records may be changed, a call of put_rec_i will ensure that the block containing the current available record will be written back before a new block is read.

Transitions between these three modes are performed explicitly by a call of the procedures set_read_i, set_update_i, or set_put_i. Such a call is also used to terminate the initialization or as preparation for close, see below.

5.5 Closing 5.5

After updating, a call of one of the mode-changing procedures, set_read_i, set_update_i, or set_put_i will ensure that all relevant information is present on the backing storage. The update mark in the filehead, however, can only be removed by set_read_i, which must be called before the file can be closed by a call of the normal RC8000 ALGOL procedure, close.

5.6 Zone State 5.6

As the file_i procs assume a specific contents of the zone buffer and the share descriptors, the zone should not be used by any procedure outside this system. The following five consecutive values of zone state are therefore reserved to describe a zone when it is used by the file_i procs:

f0+0, read_only_i:	In read_only_mode, except after call of next_rec_i.
+1, read_next_i:	In read_only_mode, after call of next_rec_i.
+2, put_i:	In put_mode.
+3, update_i:	In update_mode.
+4, initialize_i:	After call of init_file_i or init_rec_i.

The zone state is checked by all the file_i procs and an illegal value will terminate the run with an error message.
At present f0 = 10.

5.7 Results 5.7

The result of a call of a file_i proc is an integer, delivered in the standard integer variable result_i, and a zone record, the available record.

5.7.1 resulti 5.7.1

The value of result_i after a call tells about the overall result of the call; e.g. whether or not a search for a record succeeded, that the end of the file has been reached, that the record in the call has an improper length field value.

The possible values of result_i and their meanings are listed in the specification for each procedure. These values are, for each procedure, in the range from one and upwards; this makes it easy to switch on result_i or to use it in a case statement.

5.7.2 Available Record 5.7.2

During record processing there will always be an available record upon return from the file_i procs. To achieve this the file must always contain at least one record and it will be regarded as cyclic; i.e. a 'wrap-around' will be performed at the end of the file.

The available record is a normal zone record and has not been copied from the block buffer. The system relies, however, on the key- and length-fields of the record and therefore saves these before exit and restores them at the next entry; a disastrous effect of an accidental change of these fields is thus avoided.

The effect of changes made in the user fields between calls depends both on the current mode and on how the records happen to be stored in the blocks:

Let a program perform the following sequence of operations on two records, A and B:

```
get_rec_i (z, A); comment yields an available record, oldA;
change some user fields in the available record giving newA;
get_rec_i (z, B);
get_rec_i (z, A);
```

If A and B happen to be in the same block then the last operation will always yield the changed version of A, i.e. newA.
If A and B are in different blocks then the last operation will yield oldA if we are in readonly-mode or in put-mode but newA if we are in update-mode, because only in the last case will the block containing A have been written when B was accessed.

Another example, this time in put-mode:

```
get_rec_i (z, A); comment yields oldA;
change available record yielding newA;
pur_rec_i (z);
change available record yielding newnewA;
```

As the block is written when a new block is wanted the put_rec_i will include any changes made to the block from it was read-in to a new block is needed; i.e. newnewA will be the latest version of A even though it comes after the put.

In view of the uncontrolable side effects illustrated by the above examples the following rule should be obeyed.

5.7.3 Rule for Record-Updating 5.7.3

A nice program will only change the contents of the user fields in a record and only in update-mode or put-mode and only when the new version may go out to the file.

5.8 File Status 5.8

The file head and the bucket table head contain several parameters which describe the overall status of the file; e.g. noofrecs, recbytes, and transports, which is a counter holding the number of input-output operations perfomed. There are also a few parameters which it is meaningful to change; e.g. the pricelist (see insert_rec_i).

In principle the normal get_zone - set_zone mechanisms could be used to inspect, and even change, any parts of the zone buffer. For safety-reasons these mechanisms should not be used. The system therefore provides two procedures, get_params_i and set_params_i, which allow parts of the zone buffer to be inspected and selected parts to be changed (see these procedures for further details).

5.9 Error Handling 5.9

The different kinds of errors and other abnormal situations are treated as follows.

5.9.1 Input-Output 5.9.1

All transports to and from the document are initiated by explicit send-message, but they are waited for and checked by the check routine in the normal ALGOL running system. Errors and abnormal situations concerning the document are therefore handled as for any other standard input-output, i.e. the block-procedure of the zone and the giveup-mask of the open call have their usual meaning.

Output operations are normally not performed before a new contents of a buffer are needed. Whenever the system decides that a buffer has to be written before a new read is performed, it notes this by setting a write-operation in the corresponding share. In an emergency situation, e.g. an unexpected termination of the run, the file may therefore be in a bad shape. If the pending write-operations somehow, e.g. by analysis of a core-dump, can be performed, this may repair the situation. The system contains, however, no facilities for this.

5.9.2 Programming Errors 5.9.2

Logical errors, e.g. a wrong zone state at a procedure call, are treated as programming errors and will terminate the run with a run time alarm.

The possible messages are listed in A3 and they may occur if the requirements specified for each procedure are not fulfilled when that procedure is called.

5.9.3 Data Errors 5.9.3

Errors in record formats and other abnormal situations arising from the data may be detected by inspection of the result_i value upon return from a procedure call.

The user may also define that specific result_i values from specific file_i procs should invoke a call of a user specified procedure just before the file_i proc returns to the main program (see section 6.15 for further details).

6. PROCEDURE SPECIFICATIONS 6.

This chapter contains, in alphabetic order, the specifications of all the procedures offered by the system. To each file processing procedure is assigned a number, procno_i, by which the procedure is identified in the use of the test facilities (see section 6.15).

A survey of the procedures, in procno_i order, is given in Appendix A together with the possible result_i values, their meaning, and the corresponding values of available record.

6.1 Integer Procedure buflengthi 6.1

Call: buflength_i (filename, full_insert)

buflength_i (return value, integer). Number of doubleworditems needed in the zone buffer for processing the indexed-sequential file given by filename.

filename (call value, string). The name of a backing storage area containing an indexed-sequential file.

full_insert (call value, boolean). True if a buffer with room for general insertions is wanted.

Function: Reads the first segments of the document given by filename into a local zone and computes the needed buflength. The area is not released.

Errors: Uses stderror and giveup = 0. If the needed parameters in the file head do not conform to an indexed-sequential file buflength_i will yield the value zero.

6.2 Procedure deletereci 6.2

Call: delete_rec_i (z)

z (call and return value, zone).
Specifies the file.

Function: Deletes the available record from the file and makes the successor available.

Requirements: zonestate = update_i or put_i.

Results: zonestate: unchanged.
procno_i : 9

result_i :	Available record:
1 Deleted	The successor to the available.
2 Deleted, end of file	The first in the file.
3 Not deleted, only one record left	The one.

6.3 Integer Procedure getparamsi 6.3

Call: get_params_i (z) One or more pairs:(paramno, val)

get_params_i (return value, integer). Overall result of call:
0 : All parameters processed.
> 0: Exit on error in parameter pair number get_params_i.

z (call value, zone). Specifies the file.

paramno (call value, integer). Identifies the wanted value.

val (return value, integer). Receives the value identified by paramno.

Function: Yields the values of a selected set of parameters from the zone buffer of an indexed-sequential file.
The possible values of paramno and their meanings are listed in Appendix B.

Requirements: zone state = any file_i state.

Results: No change of the file.
procno_i : 12.

6 Procedure getreci 6.4

Call: get_rec_i (z, key)

z (call and return value, zone).
Specifies the file.

key (call value, real array). A record, at least up to and including all the key fields, with the same key as the one to search, i.e. key fields in the same positions as in the records with lexicographical index 1 as the base.

Function: Searches a record with the specified key and makes it available.

Requirements: zonestate = read_only_i, read_next_i, update_i, or put_i.

Results: zonestate: if zonestate = readnext_i then read_only_i else unchanged.
procno_i: 7

result_i:	Available record:
1 Found	The found.
2 Not found	The successor to the specified.
3 Not found, end of file	The first in the file.

6.5 Procedure headfilei

Call: head_file_i (z, recdescr, nkey, maxreclength, maxbucks, segsperbuck, segsperblock)

z (call and return value, zone). Specifies the document to which the generated head is output.

recdesr (call value, integer array). A two-dimensional array specifying the types and relative positions of the key- and length-fields of records.

nkey (call value, integer). The number of key fields in records.

maxreclength (call value, integer). The maximum number of doubleword items in a record. (0 < maxreclength <= 2500).

maxbucks (call value, integer). The maximum number of buckets to provide for in the bucket table of the final file. (0 < maxbucks <= 10000).

segsperbuck (call value, integer). The number of segments in a bucket in the file. Includes the segments for the block table. (1 < segsperbuck <= 1000).

segsperblock (call value, integer). The number og segments in a block in the file. (0 < segsperblock <= 50).

Function: Generates the head of an indexed-sequential file and a bucket table describing an empty file and outputs it to the document connected with z.

6.5.1 The Zone and the Document

The zone must be open. Only one share is needed, but it should be able to hold at least nkey * 10 + 45 double-words as one record in an integral number of segments. Note that this zone needs not have anything to do with the zone in which the created file later is processed.

The document will be positioned at 0, 0 and the generated file head will be output as at most two blocks by means of outrec.

The contents of the file head are independent of the document to which it is output. It may be copied to any number of documents and thus be used as head of different files which use identical record formats and block- and bucket-structure.

6.5.2 recdescr, nkey, and maxreclength 6.5.2

The array recdescr is assumed to be declared as:

```
integer array recdescr (1:nkey+1, 1:2)
```

Each of the first nkey rows describes one key field and row nkey + 1 describes the length field. The first column holds the field types and the last column the relative positions coded with the values described in section 2.1. If we have l = maxreclength * 4 then only the following relative positions are legal:

type:	relative position:
± 1	1,2,3,...,l-1,l
± 2	2,4,6,...,l-2,l
± 3	4,6,8,...,l-2,l
± 4	4,6,8,...,l-2,l

Constant length records are coded by recdescr(nkey+1, 1) = 0 and recdescr(nkey+1, 2) = anything. The record length is then assumed to be maxreclength.

6.5.2.1 Example 6.5.2.1

The record in the example in section 2.1 may be described by

```
nkey:= 3;
recdescr(1,1):=  4;   recdescr(1,2):= 10;
recdescr(2,1):= -2;   recdescr(2,2):=  2;
recdescr(3,1):= -1;   recdescr(3,2):=  5;
recdescr(4,1):=  1;   recdescr(4,2):=  5;
```

6.5.2.2 Errors 6.5.2.2

head_file_i may terminate the run with a run time alarm.
Possible causes:

recdescr <i> Error detected during processing of field i in recdescr or, if i > 2044, key exceeds capacity of a file head, only possible for nkey > 50.

head_i p <i> Other errors in parameters to head_file_i. The value of i indicates the further cause:

1 Block too small, must at least be able to hold two records of maxlength.

2 Bucket too small, already the first bucket must hold at least one block.

0 Other errors, normally absurd, e.g. negative parameters.

6.6 Procedure initfilei 6.6

Call: init_file_i (z, buckfactor, blockfactor)

z (call and return value, zone). Specifies the file.

buckfactor (call value, real). The number of blocks, useblocks, to be used in each bucket during initialization is given by: useblocks = buckfactor * blocksperbuck.

blockfactor (call value, real). The number of bytes, usebytes, to be used in each used block during initialization is given by: usebytes = blockfactor * segsize * segsperblock.

Function: Prepares an indexed-sequential file for initialization.

Requirements: zonestate = 0 after opening of an indexed-sequential file which may be empty or non-empty. The zone must have three shares and a sufficiently large buffer (see section 5.1).

Results: zonestate: initialize_i, i.e. ready for init_rec_i.

procno_i: 1

result_i:	Available record:
1 Ready	None
2 Ready, only room for simple insertions in the zone buffer	None

6.7 Procedure initreci 6.7

Call: init_rec_i (z, record)

z (call and return value, zone). Specifies the file.

record (call value, real array). Holds the record to be added from lexicographical index 1 and on.

Function: Initializes the file with the next of a sorted set of records; buckfactor and blockfactor, which have been specified to init_file_i, will determine when a new bucket or block is taken into use.

Requirements: zonestate = initialize_i after call of init_file_i or init_rec_i.

Results: zonestate: initialize_i, i.e. unchanged.

procno_i: 2

result_i:	Available record:
1 Record Added	None
2 Record not added, file is full	None
3 Record not added, improper length	None
4 Record not added, not ascending key	None

6.8 Procedure insertreci 6.8

Call: insert_rec_i (z, record)

z	(call and return value, zone). Specifies the file.
record	(call value, real array). Holds the record to be inserted from lexicographical index 1 and on.

Function: Inserts the specified record in its proper place in the file and makes it available. See below for details.

Requirements: zonestate = update_i or put_i.

Results: zonestate: unchanged

procno_i: 10

result_i:	Available record:
1 Inserted	The inserted
2 Not inserted, record with the same key already in file.	The one in the file

3	Not inserted, too expensive, can only occur with a modified insertion strategy, see below.	The successor to the specified
4	Not inserted, file is full.	The successor to the specified
5	Not inserted, improper length	The successor to the specified
6	Not inserted, there was no room for the record in the block to which it belonged and the zone buffer is too small for a more complicated insertion, see below.	The successor to the specified

6.8.1 Insertion Strategy 6.8.1

If there is room for the record in the block to which it belongs, it can be inserted without further trouble; otherwise a more complicated strategy is used. This requires an extra block in the zone buffer. Unless this block is present it is therefore pure luck if the insertion succeeds.

The following describes the full insertion strategy, it may be skipped unless you want to modify it.

The organization of the file requires that records are stored in keyorder. This means that the insertion of a new record in general will involve a reorganization of some parts of the file in order to get room for the record in the proper block.

The cost of an insertion, in terms of segment transports and other use of resources, depends strongly on how this reorganization is done. The insertion algorithm implements the following scheme which, by taking prices imposed on the involved resources into account, tries to strike a reasonable balance between a fully automatic and a user controlled strategy.

The file head holds a list of relative prices imposed on resources and with initial values assigned by head_file_i:

Name, initial value:	Meaning:
emptybuckprice,	The value of having an empty bucket.
emptyblockprice,	The value of having an empty block.
compressprice,	The initial cost of compressing, i.e. of the pushing together of records in consecutive blocks.
priceperblock,	The cost of (two block transports + central processor time) for one block involved in compressing.
priceperbuck,	The cost of (two block transports + two block table transports + central processor time) for moving an empty block over one bucket.
pricelimit,	The maximum price accepted for an insertion. If the total cost, as computed below, exceeds pricelimit then the insertion will not be done.

These prices are used to compute the total cost of an insertion in step 2, 3, and 4 of the following 7 steps which the algorithm goes through:

1: There is room for the record in the block in which it belongs: The insertion is done without further analysis.
Otherwise the insertion will push one or more records out of the block and thus create an overflow, and:

2: A pushing together of records in at most n (key-) consecutive blocks will absorb the overflow:
cost: n * priceperblock + compressprice.
and/or:

3: An empty block, not more than n buckets removed from the current, can be inserted in the block table after the current block and can thus absorb the overflow:
cost: n * priceperbuck + emptyblockprice; n may be zero;
and/or:

4: An empty bucket can be inserted in the bucket table and a block from this bucket used as in 3:
cost: emptybuckprice;
or:

5: None of the situations 2, 3, or 4 exists: The insertion is not possible, the file is regarded as full,
exit with result_i = 4;

6: None of the costs computed in step 2, 3, or 4 are less than pricelimit: The insertion is too expensive,
exit with result_i = 3;

7: The insertion is possible and is done according to the smallest cost;
exit with result_i = 1.

6.8.1.1 Changing the Strategy 6.8.1.1

A call of set_params_i can be used to set new values in the pricelist. The strategy can thereby be modified within the limits imposed by the above algorithm.

6.8.2 Example 6.8.2

Let us assume that we want to insert a whole bunch of, say, 'Jensens' in a file which is sorted according to last and first name. It may then be useful to force the system to take an empty bucket into use immediately, instead of wasting time on a more and more time consuming compressing. This can be done by assigning a low value to empty buckprice and a high value to compress price.

6.8.3 Example 6.8.3

In an on-line system it may be necessary to reject insertions which are too time consuming. This can be done by assigning a proper value to pricelimit. The number of rejected insertions may be counted and be used to indicate when a reorganization of the total file is required.

6.9 Procedure nextreci 6.9

Call: next_rec_i (z)

z (call and return value). Specifies the file.

Function: Makes the next record available.

Requirements: zonestate = read_only_i, read_next_i, update_i, or put_i.

Results: zonestate: if zonestate = readonly_i then readnext_i else unchanged

procno_i: 8

result_i:	Available record:
1 Found	The successor to the available.
2 Found, end of file	The first in the file.

6.10 Procedure putreci 6.10

Call: put_rec_i (z)

z (call and return value, zone). Specifies the file.

Functions: Notes that the current block, i.e. the block containing the currently available record, must be written back to the document before a new block is read or the mode is changed.

Requirements: zonestate = update_i or put_i.

Results:

zonestate:	unchanged		
procno_i:	11		
result_i:			Available record:
1	Done		Unchanged

6.11 Integer resulti 6.11

Yields the result of the latest call of one of the processing procedures (see Appendix A.2).

6.12 Integer Procedure setparamsi 6.12

Call: set_params_i (z) One or more pairs:(paramno, val)

set_params_i (return value, integer). Overall result of the call:

0: All parameters processed.

> 0: Exit on error in parameter pair number set_params_i.

z (call and return value, zone). Specifies the file.

paramno (call value, integer). Identifies the parameter in the zone buffer to which val is assigned.

val (call value, integer). The value to be assigned to the parameter identified by paramno.

Function: Assigns values to a selected set of parameters in the zone buffer of an indexed-sequential file. The possible values of paramno and their meanings are listed in Appendix B.

Requirements: zonestate = any file_i state.

Results: Affects only the parameters assigned to.

procno_i: 13

6.13 Procedure setputi 6.13

Call: set_put_i (z)

z (call and return value, zone). Specifies the file.

Function: Terminates the current mode and sets put-mode.

Requirements: zonestate = any file_i state.

Results: zonestate: put_i.

procno_i: 5

result_i:	Available record:
1 Normal mode change	Unchanged.
2 Initialization terminated	The first in the file.

6.14 Procedure setreadi 6.14

Call: set_read_i (z)

z (call and return value, zone). Specifies the file.

Function: Terminates the current mode and sets readonly-mode.

Requirements: zonestate = any file_i state.

Results:	zonestate:	read_only_i	
	procno_i:	4	
	result_i:		Available record:
	1	Normal mode change	Unchanged.
	2	Initialization terminated	The first in the file.

6.15 Integer Procedure settesti 6.15

Call: set_test_i (z) Optional parameter:(test_proc) one or more pairs:(procno_i, results)

set_test_i (return value, integer). Overall result of call:
- 1: Exit on error in first parameter.
0: All parameters processed.
> 0: Exit on error in parameter pair number set_test_i.

z (call and return value, zone). Specifies the file.

test_proc (call value, procedure). The name of a procedure which must be declared at the same level as the zone or at an outer level.

It must conform to the declaration:

procedure test_proc (z, record,
    procno_i); zone z; array record;
    integer procno_i;

It will, when specified, see below, be called just before the exit from a file_i proc with the following parameters:

z:	The zone of the file_i proc call.
record:	The array of the file_i proc call or, if not present, the zone z.

procno_i: The identification of the file_i proc.

The parameter test_proc may be left out if it already has been given in a previous call of set_test_i.

procno_i (call value, integer). Specifies the result_i values for which test_proc should be called upon exit from the file_i proc identified by procno_i. Any number of result_i values can be specified in one parameter by representing each result_i value as one digit in the decimal representation of results.

Function: Specifies a procedure to be called upon exit from certain file_i procs with certain result_i values.

The parameter pairs, procno_i - results, are processed in order and only specified changes in the situation will be effectuated but with the following additional conventions:

```
procno_i = 0 denotes all file_i procs.
results  = 0 denotes clearing of all previously specified
           result_i values for procno_i.
Non-existing result_i values are ignored.
```

Requirements: zonestate = any file_i state.

Results: Affects only the test situation.
procno_i: 14

6.15.1 Examples 6.15.1

The call
set test_i (z, 0, 0)
will prevent any further calls of the current test_proc.

The call

set_test_i (z, testit, 0, 123456)

will ensure that the procedure testit will be called upon exit from any file_i proc with any result_i and thus provide a means for supervising the main program.

The call

set_test_i (z, through, 0, 0, 8, 2)

will invoke a call of the procedure through when, and only when next_rec_i has reached the end of the file.
next_rec_i, procno_i = 8, yields result_i = 2 at end of file.

6.16 Procedure setupdatei 6.16

Call: set_update_i (z)

z (call and return value, zone). Specifies the file.

Function: Terminates the current mode and sets update-mode.

Requirements: zonestate = any file_i state.

Results:

zonestate: update_i.
procno_i: 6

result_i:	Available record:
1 Normal mode change	Unchanged.
2 Initialization terminated	The first in the file.

6.17 Procedure startfilei 6.17

Call: start_file_i (z)

z (call and return value, zone). Specifies the file.

Function: Prepares an indexed-sequential file for record processing.

Requirements: zonestate = 0 after opening of an indexed-sequential file containing at least one record.
The document must hold at least the same number of buckets as was used last time the file was open, it may hold more.
The zone must have three shares and a sufficiently large buffer (see chapter 3).

Results: zonestate: readonly_i, i.e. readonly-mode.
procno_i: 3

result_i:		Available record:
1	Record available	The first in the file.
2	Record available, only room for simple insertions in the zone buffer	The first in the file.
3	As 1, but updatemark found	The first in the file.
4	As 2, but updatemark found	The first in the file.

Note on resulti = 3 or 4: These results are implemented in version 12, April 1979. The file may be accessed only with zonestates readonly_i or next_i.

A. SURVEY OF THE PROCEDURES OFFERED BY THE SYSTEM A.

A.1 For Creation and Opening of an Indexed-Sequential File A.1

head_file_i (see section 6.5). External procedure which generates a file head.

buflength_i (see section 6.1) External procedure which yields the buffer size needed for processing a file.

A.2 For Processing an Indexed-Sequential File A.2

Each procedure is described below in order of their identification number, procno_i, and with possible values of result_i and available record.

procno_i, name	result_i value and meaning	Available record
1, init_file_i	1 Ready 2 Ready, short buffer	None None
2, init_rec_i	1 Record added 2 File is full 3 Improper length 4 Not ascending key	None None None None
3, start_file_i	1 Ready 2 Ready, short buffer 3 As 1, but updatemark found 4 As 2, but updatemark found	First in file First in file First in file First in file
4, set_read_i	1 OK 2 OK, after initialization	Unchanged First in file
5, set_put_i	1 OK 2 OK, after initialization	Unchanged First in file

procno_i, name	result_i value and meaning	Available record
6, set_~~i~~update_i	1 OK	Unchanged
	2 OK, after initialization	First in file
7, get_rec_i	1 Found	The found
	2 Not found	The successor
	3 Not found, end of file	First in file
8, next_rec_i	1 Found	The next in file
	2 Found, end of file	First in file
9, delete_rec_i	1 Deleted	The next in the file
	2 Deleted, end of file	First in file
	3 Not deleted, one record left	The one left
10, insert_rec_i	1 Inserted	The inserted
	2 Already in file	The one in the file
	3 Too expensive	The successor
	4 File is full	The successor
	5 Improper length	The successor
	6 Short buffer	The successor
11, put_rec_i	1 Done	Unchanged

The following utility procedures do not change result_i or available record and they cannot invoke a call of the test_proc:

12, get_params_i

13, set_params_i

14, set_test_i

A.3 Alphabetic List of Alarm Causes A.3

The system adds the messages below to the list of possible alarm causes from the standard procedures of RC8000 ALGOL.

head i p <i> Parameter error in call of head_file_i:
i = 1: Not room for two records in a block.
2: Not room for at least one block in the first bucket.
0: Other illegal parameter values.

prep i <i> Error during init_file_i, init_rec_i, or start_file_i:
i = 1: Too few or many segments in the document.
2: The bucket head is not consistent.
3: Too small a zone buffer.
4: The file head is not consistent.
5: Not three shares.
6: Zone state <> 0.
7: Empty file after start_file_i or mode change.
8: Contents field of catalog entry <> 22.
9: Updatemark found.

recdescr <i> Error or inconsistency in the record description in the call of head_file_i.
i < 2044: Error in field i.
i >= 2044: Key too big.

state i <i> Zonestate error in call of any file_i proc:
i = zonestate * 100 + procno_i.

B. PARAMETERS IN THE ZONE BUFFER B.

The lists below define the values of paramno to be used in calls of get_params_i or set_params_i.

The lists may be extended when it appears that more parameters are of interest to the user.

B.1 Parameter Values to getparamsi B.1

paramno	name	meaning
1	recsinfile	number of records in the file
2	recbytes	number of halfwords used for records
3	transports	number of input or output operations performed since the processing was started
4	pricelimit	for 4-9, see section 6.8, insert_rec_i
5	emptybuckprice	
6	emptyblockprice	
7	compressprice	
8	priceperblock	
9	priceperbuck	
10	computed cost	the cost computed in the last call of insert_rec_i

B.2 Parameter Values to setparamsi B.2

The following of the parameters above may also be assigned to by set_params_i with values in the intervals shown:

paramno	name	legal values
4	pricelimit	0 <= val <= upper limit for integers
5	emptybuckprice	0 <= val < 2048
6	emptyblockprice	0 <= val < 2048
7	compressprice	0 <= val < 2048
8	priceperblock	0 <= val < 2048
9	priceperbuck	0 <= val < 2048

RETURN LETTER

Title: RC8000 Indexed Sequential Files RCSL No.: 31-D600

A/S Regnecentralen af 1979/RC Computer A/S maintains a continual effort to improve the quality and usefulness of its publications. To do this effectively we need user feedback, your critical evaluation of this manual.

Please comment on this manual's completeness, accuracy, organization, usability, and readability:

____

____

____

____

Do you find errors in this manual? If so, specify by page.

____

____

____

____

How can this manual be improved?

____

____

____

____

Other comments?

____

____

____

____

____

Name: ____________________ Title: ____________________

Company: ____________________________________

Address: ____________________________________

Date: __________

Thank you

. . . . Fold here . . . .

. . . . Do not tear - Fold here and staple . . . .

Affix
postage
here

A/S REGNECENTRALEN
af 1979

Information Department
Lautrupbjerg 1
DK-2750 Ballerup
Denmark