Title:

Connected Files System
Users Manual

A/S REGNECENTRALEN

RC SYSTEM LIBRARY: FALKONERALLE 1 DK-2000 COPENHAGEN F

RCSL No: 28-D5
Edition: May 1972
Author: Inge Borch
Edith Rosenberg
Jørgen Winther

Keywords:

RC 4000, Software, Algol, Fortran, Procedures, Disc, Indexed Sequential Files, List Files, Chains

Abstract:

The system is a set of procedures, which can set up and process two kinds of backing store files. Records are accessed either by logical key or by chain.
98 pages.

Printed by A/S Regnecentralen, Copenhagen

Content:

	page
Introduction	2
Masterfiles	4
Listfiles	5
Chains	9
File configurations	12
Protection of files	15
Format of procedure descriptions	18
Procedure descriptions	19
Reorganization of files	50
Reorganization procedures	53
Appendix A: Survey of alarms	57
Appendix B: - - procedures	64
Appendix C: - - cf-states	67
Appendix D: Format of array chains	69
Appendix E: - - description files	70
Appendix F: - - extracted record	71
Appendix G: Zone bufferlength	73
Appendix H: Programming example	74
Appendix J: How to dimension the files	88
Appendix Z: List of keywords	94

Introduction

The connected-files-system is a set of RC 4000 Algol standard procedures, designed to handle records and links between records in files with direct access.

The system has been planned mainly to suit ordinary administrative information systems such as production- , purchase- , and sales-control, but the structures, which may be defined and processed by the system, are so general that other applications should be possible.

The central feature of the system is the chaining of records, i.e. one record holds a pointer to a next-in-chain-record. All records in one chain will have one common property, namely the starting point, which is a special record, the mother-record of the chain. Thus, when several chains passes through one record, this record will mark a link between the properties specified by the mother-records of the coinciding chains.

Chains and two kinds of files are used by the system to achieve direct access to records by key or by chain.

Master-files:

A record in a master-file is addressed by a user-defined logical key carried in the record. The master-files are organized indexed-sequentially so that fast sequential processing can be anticipated. Variable record-length is possible.

List-files:

A record is addressed by its record-number which is selected by the system at the insertion-time. The record-numbers are used internally for the chaining pointers. The physical block, in which a record is stored, is calculated directly from the record-number, but the placement of the record inside the block is read in a record-table heading the block to allow for records of variable length.

A list-file record will always be a daughter-record of one or more chains, but it may also be the mother-record of any number of chains, whereas master-file records are used as mother-records exclusively.

Chains:

A chain will always start at one record (the mother-record) in a mother-file (file holding mother-records, master- or list-file) and continue in one or more records (daughter-records) in a daughter-file (file holding daughter-records, always a list-file).

Any actual chain (string of logically connected records) in the system will belong to a certain predefined logical group of actual chains.

Depending on the context, the concept of a chain will be used in the following to denote one actual chain or a whole group

of actual chains.

A group of actual chains is characterized by the two files concerned and by the position of the record-fields used for chaining (two files may be connected by any number of chains).

The chain-groups are numbered within an actual file-configuration from one and up, the chain-number being used for initialization of the processing of that particular group of chains.

An actual file-configuration set up and processed by the connected-file-procedures may contain any number of files and any number of chains connecting them. The only limitation is that loops in the structure are forbidden, i.e. it must not be possible to meet the same file twice by stepping over the mother-file to daughter-file connections.

All files are RC 4000 backing store areas, which must be treated separately with regards to initialization, opening, closure and dumping. The connections are only checked when chains are processed and when it is attempted to delete records in mother-files.

Any file may be prolonged to a certain predefined limit in order to accommodate more data, but the best distribution of records will most likely be obtained if this facility is used sparingly.

An algol-zone is connected to each file used by a certain program, and records appear as zone-records so that no superfluous core-store-copying is performed. The record-fields used for the administration of chains are not accessible from the users program.

Master_files

These files contain the records which should be accessed directly through user-defined keywords carried in the records.

The basic file-administration-system is the Indexed-Sequential-File-System of RC 4000 described in RCSL No.55-D99. As regards the fundamental properties of the master-files, this manual should be consulted, since the corresponding procedures of the cf-system only provides for the administration of the chain-fields.
In fact, it is possible to process a single master-file by the indexed-sequential procedures, if these procedures are used exclusively, but it is not recommended, since the chain-fields of the cf-system will not be protected, and it will be possible to delete a mother-record without deleting the corresponding daughter-records.

One major difference between the set of procedures described in RCSL No.55-D99 and the cf-procedures is the way in which the files are opened and closed. In connection with the cf-system, the standard open- and close-procedures of RC 4000 algol are never used, and it is not necessary to call one of the mode-changing procedures to ensure that the file is properly updated.
File-initialization can only be terminated by the close-cf procedure and not by the mode-changing procedures.
Another difference is that the delete-m procedure has long-range effects, as it also deletes all list-file records connected to chains originating in the master-file-record.

An opened master-file is protected against unauthorized input-output-procedures by means of special zone-state values. In principle, these zone-states are just a parallel displacement of the zone-states used by the indexed-sequential procedures, see appendix C.

The cf-system will, in some cases, reference master-file-records internally for the updating of chain-fields. The logical key and not some physical address is used in this case, too, primarily because the physical location of a master-file-record may change, due to insertions and deletions, but also because the reference by key is standard, and makes it possible to reorganize a master-file without touching any other file in a file-configuration. The costs are that the chain-fields used for mother-reference are rather long, and that reference to the mother-record of a chain will require the usual search in the tables of the indexed-sequential file.

The master-record reference, which is carried in chain-fields and in the chain-tables (see chains), is a copy of the keypart described in RCSL No.55-D99, i.e. a data-field holding the keywords of a master-file-record in a compressed form.

List_files

This kind of file is designed especially for the cf-system and is intended to hold the records which only should be accessed through links from other records.
The file-administration facilitates access to a certain record in one step by a short address, as opposed to the master-files, where long key-fields and access in two steps, block-table and block, is the rule.

A record is identified by a record-number, a positive integer not greater than a maximum number determined by the size of the file and the range of record-numbers allocated to one block (The greatest possible record-number is 8.388.606).
It is not possible for the user to insert a record at a certain record-number, the file-administration itself will find an unused record-number according to a certain strategy, insert the record at this number, and insert the record-number as a 24-bit integer in the chain-field of the record which is prior-in-chain to the inserted.

File-structure:

The list-file is stored as a backing store area containing a file-head, a block-table, and a variable number of blocks.
The file-head contains information for the processing of records and the chain-tables (described under chains) of the chain-groups of which the file is the daughter. The file-head is never written back to the file.

The block-table contains a 6-bit entry per block holding a logarithmic derivative of the percentage of free room in the corresponding block. The table is held permanently in core during the processing and it is used to obtain an equal distribution of records over the whole file. This is important for the physical clustering (see insert-strategy) of records being daughters of the same chain and it limits the number of block-accesses to find a suitable block for an insertion to a maximum of two (see insert-strategy). The block-table is written back to the file together with some status-variables by the close-cf procedure or the read-only-cf procedure, if records have been removed or inserted.

A block occupies an integral number of backing store segments (1 to 8 segments, each of 512 bytes of 12 bits). All blocks are of equal length and one block corresponds to a certain user-defined range of record-numbers.
Each block is prefaced by some status-variables and a record-table of one entry per record-number allocated to that block. An entry in the record-table consists of a 12-bit byte, the rightmost bit defining, whether the record-number is free or not, and the rest, the base of the corresponding record given relative to the base of the block.

A file may be declared to hold records of either fixed or

variable length, in the latter case, the first 4 bytes of the user-part of each record is reserved by the system. The first 2 bytes will tell the length of the user-part measured in bytes and the next 2 bytes contain the record-number, both represented as integers. These 4 bytes are always restored before a new zone-record is fetched.

There are two limitations to the number of records which can be accommodated in one block: The number of record-numbers per block and the amount of room for records in one block.
In case of fixed record-length, both limitations are made equal by the cf-system, but in connection with variable length records, the user himself must balance the limitations by estimating the length of the minimum record which is going to exploit all the room of some blocks without participation of other records.
The cost of setting this minimum length too low, is one 12 bit byte per superfluous record-number, on the other hand, setting it too high, may cause some room to be left unusable in blocks mainly containing small records.

A list-file may be prolonged, but not shortened, simply by increasing the size of the backing store area of the file. This can be done during a run by use of the procedure extend-cf, or between runs by the utility-program set (System 3). However, the maximum number of blocks in one file must be given when the file-head is created, because room for a maximum block-table is reserved before the first block of the file (a block-table of one segment corresponds to 1008 blocks, two segments to 1008 + 1024, etc.).

During processing only the active part of the block-table is held in core, a large upper limit is thus not very expensive, but it is not advisable to let a list-file grow too much, and especially not too often, since this will tend to disturb the clustering of records (see insert-strategy).

File-processing:

The zone used for a list-file, holds the file-head in the first part of the zone-buffer, then the block-table as the first share of the zone, and after that a number of shares (at least one), each able to hold one block. Each block-share demands a bufferlength equal to the blocklength plus one word of 24 bits.
The use of at least two block-shares can be advantageous because return to the previously accessed block will be quite common during the maintenance of chain-fields in connection with insertion and deletion of records.
on the other hand more than one block-share can be inconvenient, as an updated block will not be written back until some other blocks have been read, so that the disc-heads have changed position.
If at least three block-shares are used, the cf-system will write back updated blocks in parallel to program execution, so

that one block-share is always ready for input with a minimum of waiting-time.
The cf-system holds a sorted list of one entry per block-share, the first entry pointing to the block which has been accessed most recently, and the last entry pointing to the victim, i.e. the block which is going to be overwritten next, because it has not been accessed for the longest period.
If the victim-block has been updated, then the transfer back to the file will be initiated, but not waited for, at the time when the block becomes the victim, provided that at least three block-shares are available.

In order to make multi-block-share runs economical and to diminish transfer-time, the user should define a small block-length, on the other hand, short blocks will demand more core for the block-table and will increase the total size of unusable block-remnants.

Insert-strategy:

The insert-strategy concerns the way in which records are placed physically in the file.
The ideal is to have records with a high probability of sequential retrieval placed sequentially or at least placed in the same neighbourhood in the file, i.e. in so few physical blocks as possible.
In a list-file, records linked logically together in the same chain will have such a great probability of sequential retrieval. It is therefore attempted to concentrate connected records physically. This will in the following be called to cluster the records.
The intention is to obtain a great probability of finding the next record of a chain in the same block as the last accessed record, the gain of having two daughter-records of one chain placed in the same block being one block-access each time the chain is traversed.
The clustering is only taken into account when a new record is going to be inserted, i.e. already placed records are never moved as that would involve very high costs.
By this simple method, it is only realistic to hope for clustering of one group of chains. A general optimization of all chains might be the task of a later, probably rather complicated, reorganization program.
Therefore, the user should favour one chain-group in each list-file by letting the insert-1 procedure work upon this chain-group, as this procedure performs the physical insertion of a list-file-record.
A list-file-record may be connected to one actual chain of each chain-group defined. One chain is connected by the insert-1 procedure, the remaining chains may be connected by the procedure connect.

The block used for the insertion is selected by insert-1 according to the following algorithm:

1. if new chain then find the block of most free room
   else
2. if room in block containing the neighbour-record
     then select this block
   else
3. if overflow has occurred earlier from the block
     containing the neighbour-record and room is
     available in the overflow-block
     then select the overflow-block
   else
4.   find the block of most free room and make this
     block the overflow-block of the block containing
     the neighbour-record.

The block containing most free room is searched in the block-table, the neighbour-record is the record which is going to be prior to the inserted record, or in case the inserted is next to the mother-record, then the record next to the inserted. The insertion is not performed in case the file is already filled beyond a user-specified percentage, or in case the block of most free room is not able to hold the record.

In case 1 of the algorithm above, an insertion will require 1 read and 1 write block-access, whereas the worst case, case 4, requires 2 read and 2 write block-accesses when at least two block-shares are available, but 4 read and 2 write block-accesses, if only one block-share is available.
When a block becomes more than half empty after a record deletion, the overflow-pointer is erased.

Zone-states:

A zone opened to a list-file may be in one of the following three states analogous to those used for the master-files:

read-only-l — It is only possible to read the file; this is the state set by the open-cf procedure. The state is not recommended, because the removal of dead records is not carried out (see chains-).

read-update-l — Both reading and writing are allowed, but to ensure that changes in a record retrieved by the procedure get-l or get-numb-l, will be reflected in the file, the procedure put-cf must be called after the retrieval. Any record read by the user may end up in the file, so the user should not make any transient changes of record-fields.

update-all-l — All accessed records are written back to the file.

Chains

A chain in the cf-system is basically a string of records, each record except the last one holding the reference to its successor.
The first record is called the mother-record and the other ones are called the daughter-records of the chain.
The mother-record and the daughter-records are placed in two separate files, called the mother-file and the daughter-file of the chain respectively.
The mother-file may be either a master-file or a list-file, but the daughter-file is always a list-file, i.e. reference to the next-in-chain record is always a list-file record-number.

All records in a chain will have a 24-bit chainfield holding either the reference to the next record in the chain or indicating end of chain, i.e. all chains are open and one-way.
In addition a daughter-record may contain a reference to the mother-record, the chain is said to be headed. This reference is either a compressed key of a master-file-record or the record-number of a list-file record. The mother-reference may be fetched by means of the procedure get-head, in order to look up the mother-record by get-m or get-numb-l, according to the type of the mother-file.
The mother-reference is intended for this purpose, which only can be of any value in case the mother-reference is wanted for a chain, different from the chain by which the record was accessed, but for the reason of security, it is checked internally that the mother-reference is the same in all daughter-records of one chain.

All records in one file will have a chain-part of the same format, each field in the chain-part corresponding to a certain chain-group, of which the file is either the mother or the daughter.
A chain-group corresponds to a certain mother-file and a certain daughter-file, and it will utilize some specific chain-fields in the records of these files.
A record in any file will contain a user-part followed by a chain-part, the user-part being of fixed or variable length and the chain-part of fixed length.

Specification:

All chain-groups in a certain file-configuration are specified by an integer array used as a parameter of the two head-procedures head-m and head-l.
A call of one of these procedures will in a backing store area generate a file-head holding among other things the specification of the chain-groups assigned to the file.
The fundamental information concerning a chain-group is the number of the chain-group (all chain-groups are numbered by the system from one and up), the position and size of the corresponding chain-fields, and the role of the file, mother or daughter.

Processing:

Before a certain group of chains can be processed, the corresponding mother-file and daughter-file must be opened and the init-chain procedure called.
This procedure will set up some absolute addresses in the zone-buffers of the two files to enable cross-reference between the two zones, and it will return a real parameter holding two absolute addresses pointing to the information in the zone-buffers concerning the chain-group.
This return value must later be used as a parameter of the various chain-processing procedures to specify the chain-group. The parameter is the one named chainref in the procedure descriptions.

Chain-tables:

The list-file zone-buffer contains a table for each chain-group of which the file is the daughter.
Each chain-table can hold the information needed to define a position in an actual chain of the corresponding chain-group.
This information consists of the following four parts:

prior — The record-number of the record which precedes the last accessed record. If the last accessed record is the first daughter-record, then prior is zero.

last accessed — The record-number of the record accessed most recently through the chain-group. It is zero if the chain-state is empty (see chain-states).

next — The record-number of the record succeeding the last accessed record. This field is copied from the next chain-field of the last accessed record.

mother — The reference to the mother-record of the actual chain stored in the same format as the corresponding record-chain-field.

The chain-tables are used by almost all procedures having a chain-parameter, the procedure get-1, for example, will use the next-field of the chain-table to retrieve the next record of a chain. The prior-field is used when the last accessed record is deleted, and when a record is connected to a chain prior to the last accessed record.

Chain-states:

A chain-group is in one of the following three states:

not-init — This is the state before the call of the procedure init-chain, but the state is also assumed when one of the two corresponding zones is closed. No chain-processing can occur in this state.

empty	The empty-state is assumed after the first call of init-chain, and in other cases specified in the procedure descriptions.
last-acc-defined	A chain-position is defined by the chain-table. The last accessed record is not necessarily the current record of the daughter-file.

Dead records:

A list-file record will always be deleted as the last accessed record of a chain-group, whether the deletion is performed explicitly by the user or internally through the file-connections.

For this chain-group it is possible to remove the record from the chain as the prior record is noted in the chain-table.

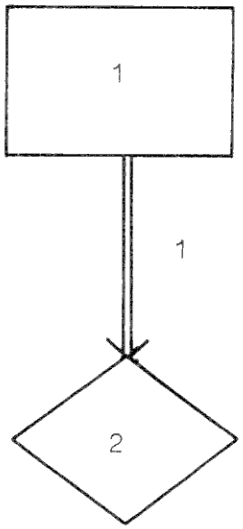
If a record is connected to one chain only, it is also removed from the file, but in the case of more than one connected chain, the record will remain in the file as a dead record until it has been disconnected from all the remaining chains. The disconnection will be performed by the system each time the dead record is retrieved as the next record of a chain, provided that the zone is in one of the update states (the mother-zone must also be in an update state if the dead record happens to be the first in the chain).

The user will thus never retrieve a deleted record, but a certain percentage of dead records in a list-file, depending on the use of the chains, must thus be taken into account. This strategy together with the use of the one-way chain has been selected to obtain a fast maintenance of chains.

File configurations

The purpose of this chapter is to propose a way of drawing diagrams defining the structure of actual file-configurations.

fig. 1



Master-file
(mother of chain 1)

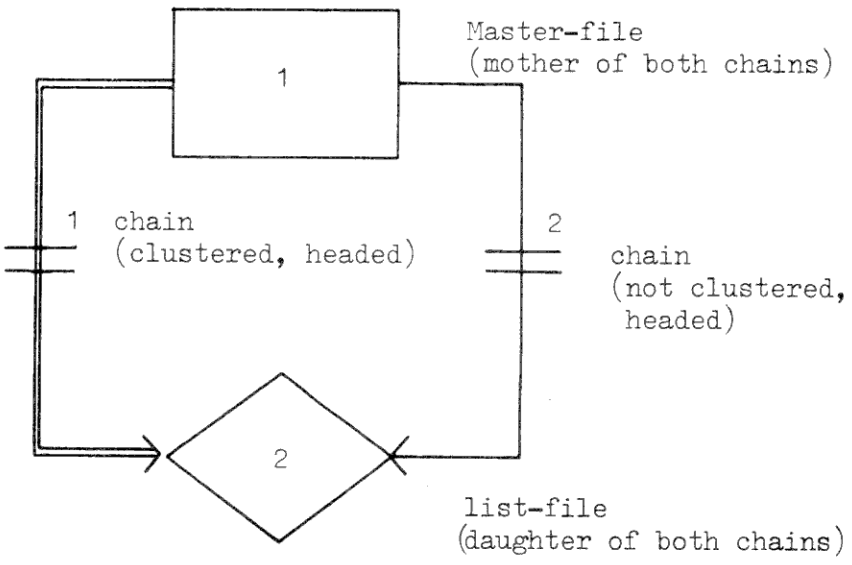
chain (clustered, no
reference to mother)

list-file
(daughter of chain 1)

The diagram of fig. 1 shows a single master-file given the logical file-number 1, a single list-file given the logical file-number 2, and a single not headed chain, chain 1 of the configuration. The double arrow is used to indicate the chain, the daughter-records of which are clustered by the insert-1 procedure, exactly one double arrow must point to a list-file.

This simple structure might be used in cases where some record-part is varying strongly in length or is infrequently used.

fig. 2

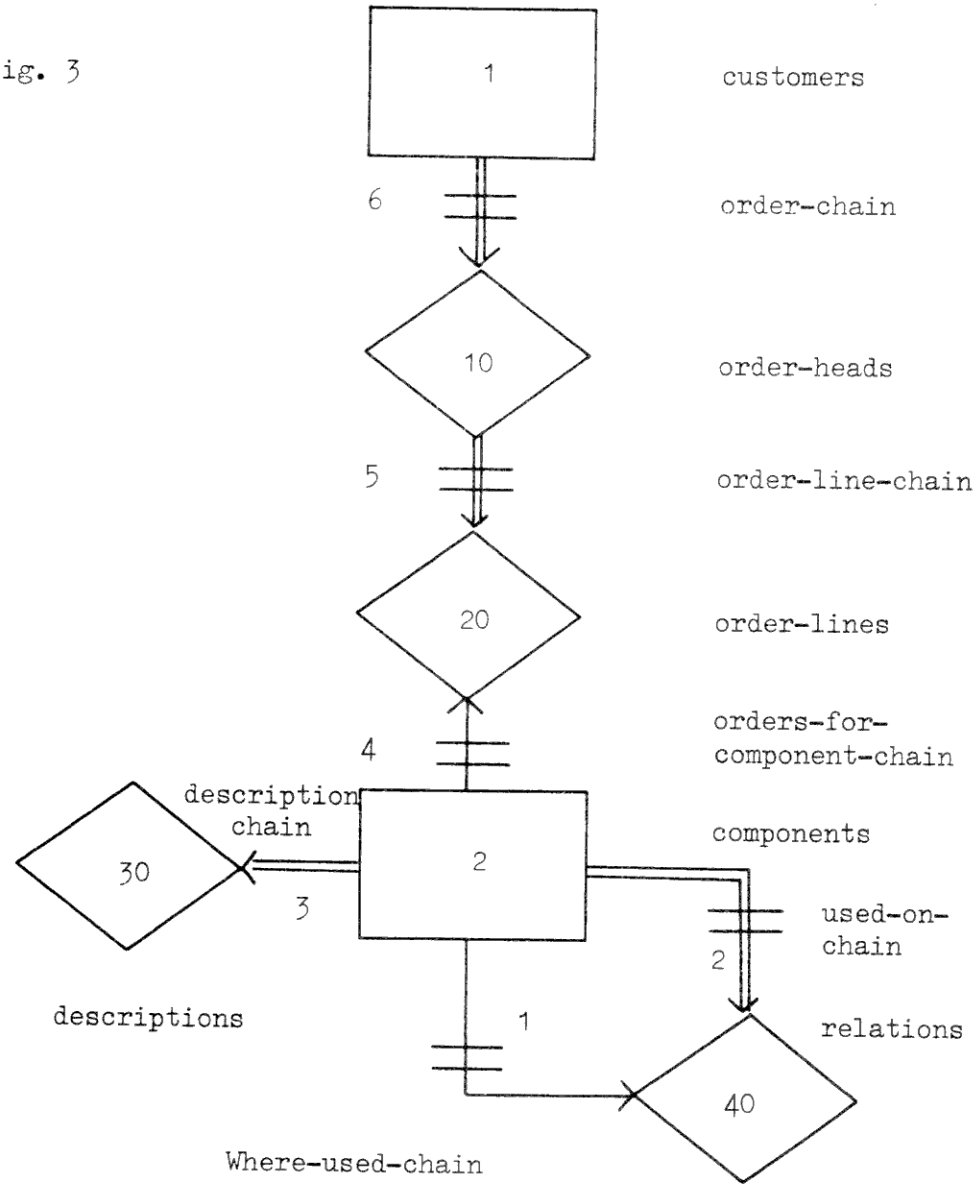


In fig. 2 the structure is extended by an extra chain-group, and both chain-groups are headed, i.e. each daughter-record holds a reference to the mother-record of the chain. This is specified by two bars crossing the arrows. Chain 1 is the clustered one.

By this configuration it is possible to look up a record in the master-file, retrieve a record of the corresponding chain 1, fetch the mother-reference of chain 2, and look up the mother-record in the master-file. Each record in the list-file may thus be thought of as a link between two records of the master-file, namely the two mother-records of the actual chains to which the list-file record is connected.

The chains of fig. 2 may, for example be used to establish the bill-of-material/where-used structure of manufactured components.

fig. 3



In fig. 3 an example is shown of the file-configuration of a sales-control system.

It may, for example, through this structure be found out, how many components are needed to effectuate the orders of one customer, or which orders have been received for a specific component.

The orders are split into two files, as one order may hold some information common to a number of order-lines, each corresponding to a certain component.

If the component-records have some lengthy and infrequently used parts, for example some text-descriptions, these parts may be stored in a separate list-file.

Protection against erroneus administration of a cf-fileconfiguration.

Error causes

An administration of the permanent files of an adp-system will face the following two error causes:

1. A file is not properly updated if the processing is not terminated by a call of some closing procedure. This call may not be executed, if a program is terminated by an operation system or by a run time alarm.

2. If more than one permanent file is used, there is a risk that different generations of files are mixed in a run. This risk is greater if a lot of files of different updating frequencies are used.

Errors of the first type may not be so serious in systems using sequential files, because the files are scanned from one end to the other, so the lack or inconsistency of some endoffile label will reveal the error.
For systems using random access files, like the cf-system, errors may remain undetected for long periods and may give rise to alarms, which are very difficult to trace back to the original cause.

The use of random access files introduces another error cause:

3. The same job or program may erroneous be run twice causing a double updating of the files involved. This is not possible in connection with sequential files, where the two runs would be completely identical, because the old versions of the files are unchanged.

Security measures in the cf-system.

The cf-system has been provided with protection against the error causes 1 and 2, but not against 3.

The catalog entry of a file is used by the protection system in this way:

file = set <segments> <bs device> <version> <update mark>.

The two last quantities are special for the cf-system:

<version> A number (0 <= version <= 8 000 000), which is increased by one each time the processing mode

is changed from read-only to update.

<update mark> Either 0 or 1.
0: The file is in read-only-state. This should be the state between runs, and this is the state accepted by open-cf.
1: The file is in update-state. This state must not occur between runs, and a file in update-state is rejected by open-cf.

Errors of type 1 are detected by means of the update-mark, which will be equal to one, if a processing in an update mode is terminated by an index alarm f.ex..

The second error type is remedied through the use of the version number in connection with a supervisory register holding the actual version numbers of all the files of a fileconfiguration. The procedure open-cf will check that the version numbers in the supervisory register, and in the catalog entry of the file are identical.

A masterfile is used as the supervisory register. It is called the description file because it can be used for all kinds of descriptions, f.ex. files, records, and fields.
A maintenance program for description files has been produced, and utility programs, and higher level cf-procedures, planned at present, will use the description file.

It is possible though, but not recommended, to switch off the version number checking. In this case the description file is not necessary, but the version number in the catalog entry is still increased.
The update mark checking can not be switched off.

The format of the description file is given in appendix E.

Set-descr-cf

This is the name of a procedure, which must be called before the first call of open-cf.
The call of set-descr-cf will provide the cf-system with the name of the relevant description file. The name is lateron used by the procedures open-cf, read-upd-cf, and update-all-cf.
If an empty string is given as the parameter of set-descr-cf, the procedures will not attempt to access a description file.

Whenever the description file is accessed, it is checked that the update mark of the description file itself is zero. Therefore the user must avoid simultaneous updating of the description file, in a zone of his own, and calls of open-cf, read-upd-cf, and update-all-cf concerning other files.

Alarms

When the protection is violated, or if the protection system has troubles with a catalog entry or the description file, a run time alarm will terminate the program. The alarm is issued by the external algol procedure protect-cf, which actually performs the functions of the protection system.

Such an alarm is headed by the following two lines:

```
xxxprotectcf  alarm:
file <file number> <file name> vers.in.cat <version>
```

The text: file, is replaced by the text: descr, if the trouble concerns the description file.

The run time alarm following these two lines will explain what happened, see appendix A, under protectcf.

Other alarms than mentioned in appendix A may arise, if the description file is not ok. The alarm will originate from either buf-length-cf or open-cf used upon the description file.

Format_of_procedure_descriptions.

<procedure name> proc.no.<proc.no.>

<abstract> (A short functional description).

Call: <procedure call> (Format of call).
<parameter description> (Explanation of each parameter).

Requirements:
(Conditions for a successful exit from the procedure. If the requirements mentioned are not fulfilled, the run will be terminated by an alarmmessage).

Results:
(Results from the procedure, inclusive notes on states and accessible records).

result_cf	current record
(possible values of the standard variable result_cf).	(specification of the accessible record, if any).

<further explanation> (Eventually some extra notes and warnings).

buf_length_cf proc.no. 1

Returns the bufferlength of a zone to be used for a connected file.

Call: buf_length_cf (filename, blocks_in_core)

buf_length_cf	(return value, integer) The needed bufferlength.
filename	(call value, string) The name of a backing store area holding a cf-file.
blocks_in_core	(call value, integer) Defines the number of blocks wanted in core at the same time: Masterfiles: blocks_in_core = 1, or 2 if full insertion is wanted. Listfiles: blocks_in_core >= 1, >= 2 is recommended.

Requirements:
filename must describe a backing store area holding a correct masterfile or listfile, and must not be reserved.

Results:
result_cf = 1, ok

Further explanation:
Declares a zone, opens the file, reads the first segments, and computes the needed bufferlength according to blocks_in_core.

A masterfile-zone may be declared:
```
zone zm(buf_length_cf(filename, 1 or 2), 3, blockproc);
```

A listfile-zone may be declared:
```
zone zl(buf_length_cf(filename, blocks_in_core),
     blocks_in_core +1, blockproc);
```

Documenterrors will cause stderror to be called.

close_cf proc.no. 6

Terminates the use of the zone by writing back eventual updated blocks.

Call: close_cf (z, rel)

z	(call and return value, zone) Connected to a masterfile or listfile.
rel	As for algol procedure close.

Requirements:
zonestate = any cf-state, exept after-declaration.

Results:
zonestate = 4, after-declaration.

result_cf = 1 ok

Chainstates will be not_init for associated chains.

connect proc.no. 13

The procedure connects the last accessed record in one chain to another chain according to a specified mode.

Call: connect (z1, chainref_1, chainref_2, icmode)

z1	(call and return value, zone) Connected to a listfile.
chainref_1	(call value, real) Return parameter from init_chain. The record to be connected is the last accessed record of this chain.
chainref_2	(call value, real) Return parameter from init_chain. The reference for the chain to connect to.
icmode	(call value, integer)
= 1	connect chain_1 record as first member of chain_2 from current record in the motherfile of chain_2.
= 2	connect chain_1 record as next to last accessed record in chain_2.
= 3	connect chain_1 record as prior to last accessed record in chain_2.

Requirements:

z1 must be daugtherfile of both chain_1 and chain_2.
zonestate = read_update_1 or update_all_1.
chainstate (chain_1) = last_accessed_def.
If icmode = 1, chainstate (chain_2) may be empty else last_accessed_def.
If icmode = 1, then current record in the motherfile must exist.
If the connected record is next to the mother record, the motherfile must be in an update state.

Results:

If result_cf = 1, the connected record will be last-accessed in chain_2, too. Zonestates are unchanged.

result_cf	current record
1 connected	the connected
2 not connected (already conn. to another chain)	none

Note: For icmode = 3: see the note for the procedure insert_1.

delete_chain proc.no. 16

The procedure deletes all records in a chain headed to current record of a file and all records in chains originating in records of the specified chain.

Call: delete_chain (z, chainref)

z	(call and return value, zone) Connected to a masterfile or listfile.
chainref	(call value, real) Return parameter from init_chain.

Requirements:

zonestate = read_update_m, update_all_m or read_update_l, update_all_l depending on the type of the specified file. All chains originating in the daughterfile given by chainref must be initialized. Current record must exist. All daughterfiles must be in an update state.

Results:

zonestate is unchanged. Chainstates become empty for all chains associated to files, where records have been deleted.

result_cf	current record
1 deleted	unchanged
2 no chain to delete	-

delete_1 proc.no. 15

The procedure deletes the last accessed record in the chain and all records in chains originating in the record. The next record in the chain becomes current record of the file.

Call: delete_1 (z1, chainref)

z1	(call and return value, zone) Connected to a listfile.
chainref	(call value, real) Return parameter from init_chain.

Requirements:
zonestate = read_update_1 or update_all_1.
chainstate = last_accessed_def.
All chains originating in the listfile must be initialized.
All daughterfiles and the motherfile corresponding to chainref must be in an update state.

Results:
zonestate is unchanged. Chainstates become empty for all chains associated to files where records have been deleted, except the chain specified as parameter (see below). Other files where records have been deleted, will have no current record.

result_cf	current record
1 deleted	the next in chain
2 - ,last in chain, chainstate = empty	none

delete_m proc.no. 14

The procedure deletes the current record of the file and all records in chains originating in the masterrecord.

Call: delete_m (zm)

zm (call and return value, zone) Connected to a masterfile.

Requirements:
zonestate = read_update_m or update_all_m. All chains originating in the masterfile must be initialized. All daughterfiles must be in an update state.

Results:
Chainstates become empty for all chaingroups associated to daughterfiles, where records have been deleted. Other files, where records have been deleted, will have no current record.

result_cf	current record
1 deleted	the next in the file
2 - ,end of file	the first
3 not deleted, only one left in the file.	the one

It is obvious that the call may have rather wide consequences. In case of several connected files it is advisable to use the procedure delete_chain in connection with delete_m to get more informative results.

extend_cf proc.no. 2

The procedure increases the length of a cf-file during the processing. The current record, zone- and chainstates are preserved.

Call: extend_cf (z, segments)

z	(call and return value, zone) Connected to a masterfile or listfile.
segments	(call value, integer) The extension in segments.

Requirements:

zonestate = read_only, read_update, or update_all, _m or _l. The zonestate is checked by a call of read_only_cf.

Segments >= 0, and not so great, that max_bucks or max_blocks is violated.
The bufferlength of the zone must be sufficient for the extended file.
Segments and bufferlength are checked in a call of open_cf performed on the extended file.

Results:

Current record, and all states are unchanged for any value of result_cf.

result_cf

1	ok
2	ok, but only room for simple insertion in masterfile buffer.
> 10000	error in a call of a monitor function. result_cf = result of monitor call × 10 000 + number of monitor function. Probable results: 40044 change-entry, the scope of the file does not permit change. 60044 change-entry, there is not room for the extension.

get_head proc.no. 10

The procedure is used on current record in a list_file to give the key of the mother-record of a chain to which the listfile-record is connected.
The key may evt. be a rec_no of a listfile record.

Call: get_head (z1, chainref, key)

z1	(call and return value, zone) Connected to a listfile
chainref	(call value, real) Return parameter from init_chain.
key	(return value, real array) See keywords, app. Z.

Requirements:
zonestate = read_only_1, read_update_1 or update_all_1.
chainstate = last_acc_def or empty, current record must exist.
z1 must be the daughter_file of the chain.

Results:

result_cf	current record
1 ok	unchanged
2 record not connected	unchanged, and key unchanged

get_l proc.no. 9

The procedure searches a new current record in a listfile.

Call: get_l(zl, chainref, gmode)

zl		(call and return value, zone) Connected to a listfile.
chainref		(call value, real) Return parameter from init_chain.
gmode		(call value, integer)
	= 1	the wanted record is the first member of the chain from current record in the motherfile.
	= 2	the wanted record is the one next to the last accessed record in the chain.
	= 3	the wanted record is the last accessed in the chain.

Requirements:
zonestate = read_only_l, read_update_l or update_all_l.
chainstate = last_accessed_def, or if gmode = 1, empty.
If gmode = 1, current record in the motherfile must exist.

Results:

result_cf	current record
1 found	the wanted
2 not found	if gmode=2 then the last accessed else none

If no current record then chainstate = empty else last accessed record corresponds to current record.

get_m proc.no. 8

The procedure searches a record in a master-file with a specified key and makes it current record.

Call: get_m(zm, key)

zm	(call and return value, zone) Connected to a masterfile.
key	(call value, real array) See keywords, app. Z.

Requirements:
zonestate = read_only_m, read_update_m, or update_all_m.

Results:

result_cf	current record
1 found	the found
2 not found	next with a greater key
3 - - , end of file	the first

get_numb_1 proc.no. 23.

The procedure makes a listfile record given by its record-number available as current record.

Call: get_numb_1 (z1, rec_no)

z1	(call and return value, zone) Connected to a listfile.
rec_no	(call value, integer) Contains the number of the wanted record.

Requirements:
zonestate = read_only_1, read_update_1 or update_all_1.

Results:

result_cf	current record
1 record active	the wanted
2 record dead	none

zonestate and chainstate are unchanged.

get_param_cf proc.no. 30

The procedure yields the values of a selected set of parameters from the zonebuffer of a cf-file.

Call: get_param_cf (z) one or more pairs: (paramno, val)

z	(call and return value, zone) Conncted to a masterfile or listfile.
paramno	(call value, integer) Identifies the wanted zoneparameter.
val	(return value, integer) Receives the value of the zoneparameter identified by paramno.

Requirements:

The zone must be opened by open_cf or init_file_m.

If the file is a masterfile, paramno must be one of the values listed in RCSL No. 55-D99, appendix B1.

If the file is a listfile, paramno can be one of the following numbers:

paramno	name	meaning
1	dead-bytes	Number of bytes occupied by dead records (including chain-parts).
2	used-bytes	Number of bytes used by records (incl. dead records).
3	fill-limit	The maximum allowed percentage of used-bytes in the file. (Standard is 80 pct. for a not empty file.)

Results:

result_cf = 1, ok.

Alarm -par.pair- occurs when an error is found in the parameter-list. Alarmno shows the number of the parameterpair, where the error was found.

head_l proc.no. 26

The procedure will generate the head of a listfile in a backing store file. (See app. J. for selection of size_l)

Call: head_l (filename, file_no, chains, size_l)

filename	see procedure head_m
file_no	- - -
chains	- - -
size_l	(call value, integer array) Contains the following 4 integers:

fixed_rec_length
= 0 means variable record length is wanted,
> 0 means fixed length is wanted, the value specifies the length in bytes.

min_rec_length in case of variable length, this integer specifies the minimum length of records, which should fill a block without participation of longer records.

segs_per_block number of segments in a block. (1<= segs_per_block <= 8). The length of the greatest record that can be inserted in a block may be calculated thus:

```
chain_part_size :=
    2×no_of_associated_chains
    + sigma(type×compressed_key_
    size) over all chains of which
    the file is the daughter;
    comment see array chains;

max_no_of_recs_per_block :=
    (512×segs_per_ block //
    (min_rec_length +
    chain_part_size + 1) + 1)
    // 2 × 2;

max_rec_length :=
    512 × segs_per_block -
    (chain_part_size +
    max_no_of_recs_per_block + 8);
```

max_blocks the maximum number of blocks the file will ever hold.

Results:

result_cf = 1 ok, file_head is created.

head_m proc.no. 25

The procedure will generate the head of a masterfile in a backing store file. (See app. J. for selection of size_m)

Call: head_m (filename, file_no, chains, rec_descr, no_of_keys, size_m)

filename (call value, string)
The name of the backing store file.

file_no (call value, integer)
The logical number of the file used in chain specifications.

chains (call and return value, integer array)
Contains the specification of all chain-groups in the system. The procedure returns the quantity compressed_key_size for the associated chains.
See format of array chains in app. D.

rec_descr (call value, integer array) A two dimensional array (1:no_of_keys+1,1:2) holding information about types and relative locations of the keywords and the length in a record.
Same conventions as in RCSL 55-D99, the length in element no_of_keys+1, with type=0 for fixed length records.

no_of_keys (call value, integer) The number of keywords.

size_m (call value, integer array)
Contains the following 4 integers:

maxreclength maximum length, in bytes, of records which will be stored in the file.

maxbucks maximum number of buckets the file will ever hold.

segsperbuck the number of segments in one bucket.

segsperblock the number of segments in one block.

Results:
result_cf = 1 ok, file_head is created.

init_chain proc. no. 5

The procedure establishes the connection between the two zones used for the motherfile and the daughterfile of a chain-group.

Call: init_chain (z, z1, chainno, chainref)

z	(call and return value, zone) Connected to a masterfile or listfile. This zone must be opened to the mother-file.
z1	(call and return value, zone) Connected to a listfile. This zone must be opened to the daughter-file.
chainno	(call value, integer) The number of the chain-group in the array chains. (See app. D).
chainref	(return value, real) This real is lateron used as chain-reference.

Requirements:

zonestate (z) = read_only_m or _l, read_update_m or _l, update_all_m or _l
zonestate (z1)= read_only_l, read_update_l, update_all_l
Chainno must describe a chaingroup connecting the two files to which z and z1 have been opened.

Results:

result_cf = 1, ok
if chainstate = not-init then chainstate = empty
else chainstate is unchanged.
chainref = chain-reference

init_file_m proc.no. 27

The procedure prepares a backing store file for initialization. The file must contain a master file head. The initialization must be effectuated by successive calls of init_rec_m and terminated by a call of close-cf.

Call: init_file_m (zm, filename, giveup, buckfactor, blockfactor)

zm	(call and return value, zone) A zone with room for at least one block (see procedure buflength_cf).
filename	(call value, string) The name of a backing store area holding a file head.
giveup	As for algol standard procedure open.
buckfactor	(call value, real) See file_i procedure init_file_i.
blockfactor	(call value, real) See file_i procedure init_file_i.

Requirements:
zonestate = 4, after declaration.
The zone must be declared with exact 3 shares, and have a sufficient large buffer area. The file must contain a correct head.

Results:
result_cf = 1 ok
zonestate = init_m.

init_rec_m proc.no. 28

The procedure is used to add records to the file one by one in the key order. All chain-fields are empty after the insertion. The initialization should be terminated by a call of close-cf.

Call: init_rec_m (zm, record)

zm	(call and return value, zone) Connected to a masterfile by init_file_m.
record	(call value, real array) The record to be inserted.

Requirements:
zonestate = init_m.

Results:

result_cf	current record
1 record added	none
2 not added, file is full	none
3 - - , improper length	none
4 - - , - key	none

insert_1 proc.no. 12

The procedure inserts a record in a chain according to a specified mode, and makes it available as the current record.

Call: insert_1 (z1, chainref, icmode, record)

z1		(call and return value, zone) Connected to a listfile.
chainref		(call value, real) Return parameter from init_chain.
icmode		(call value, integer)
	= 1	insert record as first member of the chain from current record in the motherfile.
	= 2	next to last accessed record in the chain
	= 3	prior to last accessed record in the chain.
record		(call value, real array). If variable-length the lexicographical first element must contain 0.0 shift 24 add length shift 24.

Requirements:
zonestate = read_update_1 or update_all_1.
chainstate = last_accessed_def or if icmode = 1, empty.
If icmode = 1 then current record in the motherfile must exist. If the motherfile is touched, it must be in an update state.

Results:
chainstate = last_accessed_def if result_cf = 1.

result_cf	current record
1 inserted	the inserted
2 fill limit exceeded	none
3 length error	-
4 no block can take this record	-

The users record is expanded with the necessary chainfields (all empty) before the insertion.
The inserted record will later be transferred to the file.

Note: For icmode = 3: if last accessed is next to a motherfile record, this record will be current record of the motherfile after the call.

insert_m proc.no. 11

The procedure inserts a record in the proper place in the file and makes it available as the zonerecord.

Call: insert_m (zm, record)

zm	(call and return value, zone) Connected to a masterfile.
record	(call value, real array) The record to be inserted.

Requirements:
zonestate = read_update_m or update_all_m.

Results:

result_cf	current record
1 inserted	the inserted
2 record already in file	the one in the file
3 not inserted, too expensive	next with a greater key
4 file is full	- - - - -
5 length error	- - - - -
6 no buffer	- - - - -

The users record is expanded with the necessary chainfields (all empty) before insertion.
The inserted record will later be transferred to the file.

new_recl_cf proc.no. 24

The procedure is used for changing the record-length of the current record, only possible for masterfiles with variable recordlength.

Call: new_recl_cf (zm, length)

zm	(call and return value, zone) Connected to a masterfile.
length	(call value, integer) Defines the new length in bytes.

Requirements:
zonestate = read_update_m or update_all_m.
variable record_length defined.

Results:

result_cf	current record
1 changed	the same
2 last rec. in file	same with the old length
3 too expensive	- - - - -
4 file is full	- - - - -
5 length error	- - - - -
6 no buffer	- - - - -

In case length is less than the original length, elements are squeezed out from the upper end, otherwise data are unchanged.

next_m proc.no. 17

Makes the next record in a master-file current record.

Call: next_m (zm)

zm (call and return value, zone) Connected to a masterfile.

Requirements:
zonestate = read_only_m, read_update_m or update_all_m.

Results:

result_cf	current record
1 found	the next
2 found, end of file	the first

open_cf proc.no. 3

The procedure opens the zone for the specified file and prepares it for use by the other file_cf procedures.

Call: open_cf (z, filename, giveup)

z	(call and return value, zone) A zone with room for at least one block (see procedure buflength_cf).
filename	(call value, string) The name of a backing store area holding a file head.
giveup	As for the algol standard procedure open. Yet open_cf will always set the end-of-document-bit (1 shift 18) in the give-up-mask.

Requirements:
zonestate = 4, after declaration.
Filehead ok, masterfiles must contain at least one record.
Filename must be known.
Set_descr_cf must have been called.

Results:
zonestate = if masterfile then read_only_m
else read_only_1.

result_cf	current record
1 ok	if masterfile then the first else none
2 ok, but only room for simple insertion in the masterfile zonebuffer	the first in the masterfile

If the program tries to open a file, which is not initialized, the run will be terminated by an alarm probably concerning a masterfile-error, even if the file was expected to be a listfile.

protect_cf proc.no. 33

Special purpose procedure.
The procedure is called internally by the cf-system in order to have update marks and version numbers checked. But it can be called directly if the name of the current description register is wanted by some standard procedure.

Call: protect_cf (z, action)

z	(return value, zone) Will contain the name of the description file if set_descr_cf was called before this call.
action	(call and return value, integer) Must equal -1. It is changed to 0 if set_descr_cf was not called.

Requirements:
action = -1.
Other values of action may have peculiar results.

put_cf proc.no. 18

The procedure ensures that the current record will be transferred to the file.

Call: put_cf (z)

z (call and return value, zone) Connected to a masterfile or listfile.

Requirements:
zonestate = read_update or update_all.

Results:

result_cf	current record
1 ok	unchanged

The procedure is -dummy- when zonestate = update_all, or the current record is created by insert.

read_only_cf proc.no. 19

Transfers updated blocks to the file, and sets the zonestate to read_only_m or _l.

Call: read_only_cf (z)

z (call and return value, zone) Connected to a masterfile or listfile.

Requirements:
zonestate = read_only_m or _l, read_update_m or _l, or update_all_m or _l.

Results:
result_cf = 1, ok
zonestate = read_only_m or _l.
current record is unchanged.

read_upd_cf proc.no. 20

If zonestate = read_only_m or _l and a current record exists, a new copy is transferred from the file. Zonestate is set to read_update_m or _l.

Call: read_upd_cf (z)

z (call and return value, zone) Connected to a masterfile or llstfile.

Requirements:
zonestate = read_only_m or _l, read_update_m or _l, update_all_m or _l.

Results:
result_cf = 1, ok
zonestate = read_update_m or _l.
current record is the same, but evt. a new copy from the file.

set_descr_cf proc.no. 32

This procedure must be called at least once in any program using open_cf. The call must precede the first call of open_cf. The procedure provides the cf-system with the name of a description file. The description file is accessed internally by the cf-system for checking, and updating of version numbers in the procedure open_cf, and in the procedures read_upd_cf and update_all_cf if the prior zone state was readonly.
Set_descr_cf may be called several times, if more description files are involved in a run, and the parameter of set_descr_cf may be empty, indicating that no description file should be accessed.

Call: set_descr_cf (descr_file)

descr_file (call value, string) The name of the description file, or an empty string (<::>). In the latter case the version number check is not performed.

Requirements:
The call is always legal, the existence of the descriptionfile is not checked by set_descr_cf.

set_jumps_cf proc.no. 4

The procedure specifies for a certain zone a user-procedure to be called when certain values of cf-proc-no and result-cf coincide at exit from a cf-procedure. These cases are specified by the parameter-pairs cf_proc_no and results.

Call: set_jumps_cf (z, jump_proc)
one or more pairs: (cf_proc_no, results)

z	(call and return value, zone) Connected to a masterfile or listfile.
jump_proc	(procedure) The name of the users procedure, which must be declared at the same blocklevel as the zone, or at an outer level. It should be declared thus: jump_proc (z, cf_proc_no).
cf_proc_no	(call value, integer) and
results	(call value, integer) Specifies the result_cf-values for which jump_proc should be called upon exit from the cf-procedure identified by cf_proc_no.

Requirements:
The zone must be opened by open_cf or init_file_m.

Jump_proc cannot be called from those cf-procedures which are external algol procedures (see app. B), nor from open_cf, get_param_cf, or set_param_cf. If cf_proc_no specifies one of these procedures, it will be neglected.

cf_proc_no = 0 denotes all possible cf-procedures.
results = 0 denotes clearing of all previously specified result_cf values for cf_proc_no. Non-existing result_cf values are ignored.

Any number of result_cf values can be specified in one parameter by representing each result_cf value as one digit in the decimal representation of results. As the result-digits are processed from behind, result = 120 will clear old specifications and set the new values 2 and 1.

Alarm -par.pair- occurs when an error is found in the parameterlist. An alarmno > 0 shows the number of the parameter pair, where the error was found, alarmno = 0 denotes an error in jump_proc (e.g. declared at a wrong blocklevel).

The parameter pair (1,1) needs a special explanation:

If this parameter pair has been given, the jumpproc is called as:

alarmproc (z, -cf_proc_no, alarm_number)

where alarm_number is an integer specifying the number of an alarm occurring during the processing of zone z.

If alarmproc returns through its final end, the usual alarm is given, but it is possible by a goto out of alarmproc to continue the processing.

It is only possible to trap alarms occurring when it is sure that zone z contains a correct filehead. I.e., it is not possible to trap zonestate alarms or the alarms from open_cf and init_file_m.

Alarms from procedures coded in algol cannot be trapped.

set_param_cf proc.no. 31

The procedure assigns new values to a selected set of parameters in the zonebuffer of a cf-file.

Call: set_param_cf (z) one or more pairs: (paramno, val)

z	(call and return value, zone) Connected to a masterfile or listfile.
paramno	(call value, integer) Identifies the zoneparameter to be changed.
val	(call value, integer) The new value to be assigned to the zoneparameter identified by paramno.

Requirements:

The zone must be opened by open_cf or init_file_m.

For a masterfile the allowed set of values for paramno and val is listed in RCSL No. The parameters will only be changed in the zonebuffer, but not in the file. 55-D99, appendix B2.

For a listfile the only parameter which can be changed is fill-limit, i.e. paramno = 3 (see get_param_cf), and 1 <= val <= 100. The value will be inserted in the zonebuffer as well as in the file.

Results:

result_cf = 1, ok.

Alarm -par.pair- occurs when an error is found in the parameter-list. Alarmno shows the number of the parameterpair, where the error was found.

update_all_cf proc.no. 21

If zonestate = read_only_m or _l and a current record exists, a new copy is transferred from the file. Zonestate is set to update_all_m or _l.

Call: update_all_cf (z)

z (call and return value, zone) Connected to a masterfile or listfile.

Requirements:
zonestate = read_only_m or _l, read_update_m or _l, update_all_m or _l.

Results:
result_cf = 1, ok
zonestate = update_all_m or _l.
current record is the same, but evt. a new copy from the file.

Reorganization

Normally the cf-files should be selfmaintaining, special overflow areas f.ex. are never used, and deleted records can be cleaned out during the normal use. But it may of course happen, that record formats must be changed, that new chain-groups must be created, or old ones removed, or that a new version of the cf-system demands that fileheads of the existing files are changed.

For doing this kind of reorganization, four procedures are introduced: init_extract, extract_cf, init_add, and add_cf.

The basic scheme of a file reorganization, using these procedures, is the following:

1. All records of a file are extracted one by one in sequential order. The extracted records will contain the userparts as well as the chainparts of the original records.

2. The extracted records are transformed according to the new record format. Care must be taken to preserve inter-record-references. If listfile records are renumbered or masterfile keys are changed, the corresponding references must also be changed.

3. A new filehead is created according to the new demands.

4. Records are added to the new file in sequential order. Masterfile records are added in ascending keyorder and listfile records are added at certain record-numbers, normally the same record-numbers as before the reorganization, in increasing record-number order.

The procedures init_extract and extract_cf are used in step 1 to fetch the records.

Tools for execution of step 2 are not provided here, but it should on the other hand be possible to perform step 2 in a reasonable way by means of programs coded in algol or fortran.

The procedures head_m or head_l may be used in step 3 for the creation of the new filehead, and the procedures init_add, add_cf, and close_cf are used for the reinsertion of records in step 4.

The scheme can be used for any kind of reorganization, but it should be emphasized that reorganization involving resequencing of records will be very complicated, even removal of dead records from a listfile , if done sequentially, will involve much sorting and access to the relevant motherfiles.

So, in the following, only the simple reorganization of one

file, i.e. cases, where all records of one file are extracted and added again in the same order, will be considered.

In this kind of reorganization, the keys of master records, and the keys of list records will be unchanged.

The reason for such a reorganization can be one of the following:

1. A new version of the cf-system demanding a new filehead is released.

2. You want to make a compressed dump of a file on magnetic tape without unused space and administrative tables.

3. You want to have masterfile records distributed evenly over the whole file with a certain filling factor, or you will decrease the total length of the masterfile. Listfiles cannot be shortened because the mapping of record numbers on the physical blocks is not changed.

4. Some file parameters should be changed, f.ex. segs-per-block, segs-per-bucket, max-bucks, or max-blocks.

In these four cases step 2 in the basic reorganization scheme is not needed.

5. The record format should be changed. New fields must f.ex. be added, or old ones removed, or the recordlength should be made variable, etc..

6. New chain groups should be created or old ones removed. This involves a change of the chainparts of all records of files associated with those chain groups.

Warning_concerning_listfiles_of_variable_length_records.

The simple reorganization can always be performed on masterfiles, and on listfiles of fixed length records.

But in connection with listfiles of variable length records it is not sure, that all records can be added to the new version of the file, if some recordlengths have been increased, or if the min-rec-length- or the segs-per-block-parameter of head_1 has been changed.

This problem is due to the fact that the record number of a listfile record is not changed by the reorganization.

A group of longer records, which in the old version of the file were placed in separate blocks, may happen to belong to the same block in the new version, or have grown so big, that they cannot be accomodated in the block any more.

A remedy to this, is to have a smaller quantity of record numbers per unit of physical room. This can be obtained through the use of a greater value of min-rec-length, the parameter size_1(2) of the procedure head_1. But you can normally not be sure, that all records can go into the new version of the file, and the more sparce mapping of record numbers on the physical room, will on the other hand increase the size of the listfile.

NB. The reorganization procedures are not coded yet.

init_extract proc.no. 34

Reorganization procedure.
The procedure prepares a cf-file for extraction of records.
The extraction must be effectuated by successive calls of the procedure extract_cf, and terminated by a call of close_cf.

Call: init_extract (z, filename, giveup)

z	(call and return value, zone) A zone with room for at least one block (see procedure buflength_cf).
filename	(call value, string) Name of backing store area holding a cf-file.
giveup	(call value, integer) As for algol standard procedure open.

Requirements:
zonestate = 4, after declaration.
filename must point to a backing storage area containing a cf-file.
If the file is a masterfile, it must contain at least one record.

Results:
zonestate = extract-cf.
result_cf = 1, no current record.

extract_cf proc.no. 36

Reorganization procedure.
The procedure creates an extracted record in the array given as the second parameter. Before extract_cf can be used, the procedure init_extract must have been called.
The first call of extract_cf will yield the first record of the file, the next call the next etc..
Note that also dead listfile records are extracted.
See appendix F for the format of an extracted record.

Call: extract_cf (z, extract_rec)

z	(call and return value, zone) Connected to either a masterfile or a listfile by init_extract.
extract_rec	(return value, real array or zone) Will hold the extracted record if not end of file. The record is stored from byte 1 and on.

Requirements:
zonestate = extract_cf.
result_cf = 2 must not have occurred.
The bounds of extract_rec must include the byte numbers 1 and total_length. (see appendix F).

Results:
no current record.

result_cf

1 ok
2 end of file

In case of result_cf = 2 extract_rec is unchanged, and a succeeding call of extract_cf will give an alarm.

init_add proc.no. 35

Reorganization procedure.
The procedure prepares a cf-file for addition of records.
The addition must be effectuated by successive calls of the procedure add_cf, and terminated by a call of close_cf.

Call: init_add (z, filename, buckfactor, blockfactor)

z	(call and return value, zone) A zone with room for at least one block (see procedure buflength_cf).
filename	(call value, string) Name of a backing store area holding a cf-file.
giveup	(call value, integer) As for algol standard procedure open.
buckfactor	(call value, real) If listfile then not used, if masterfile then see file_i procedure init_file_i.
blockfactor	(call value, real) See buckfactor above.

Requirements:
zonestate = 4, after declaration.
filename must point to a backing storage area holding a correct cf-filehead.

Results:
zonestate = add_cf.
result_cf = 1, no current record.

add_cf proc.no. 37

Reorganization procedure.
The procedure adds an extracted record given by the second parameter to the file given by the first parameter.
Before add_cf can be used, the procedure init_add must have been called.
The records are added in ascending key- or recordnumber-order.
See appendix F for the format of an extracted record.

Call: add_cf (z, extract_rec)

z	(call and return value, zone) Connected to either a masterfile or a listfile by init_add.
extract_rec	(call value, real array or zone) The extracted record must be stored here from byte 1 and on.

Requirements:
zonestate = add_cf.
The bounds of extract_rec must include the byte numbers 1 and total_length.
The total_length must equal 8 + user_part_size + chain_part_size. (See appendix F.)

Results:
no current record.

result_cf

1	ok
2	not added, masterfile is full, or recno too great
3	- - , improper user_part_size
4	- - , descending master key or recno.
5	- - , not room in listfile block.

Errors_during_processing_of_cf-files.

Errors may be found at several levels:

1. Standard errors, i.e. errors concerning the device and the transfers, may be analysed in the blockprocedure, as in any other algol input-output procedure. The giveup mask is a call value to the cf_procedure open_cf. However, end of document has a special treatment in the cf_system, as the masterfiles are regarded as being cyclic, and end of document in a listfile means addressing outside the area, which should be impossible. (See the procedures get_m and get_l).

2. Unnormal situations: As a general philosophy is chosen that it is not up to the cf-system to decide what may be regarded as -normal- and -unnormal-, as far as normal -bookkeeping- can be maintained. The standard integer variable result_cf will yield the result of a procedure call, which always should be checked by the user. Any result of any cf-procedure may also be caught in a procedure specified as a call value to the procedure set_jumps_cf, though its original purpose rather is to give a facility for supervision during debugging of the program.

3. Grave logical errors, i.e. requirements are not fulfilled at a procedure call, will always terminate the run with an algol run time alarm. In this case the various zones are not closed, and files which were in an update mode at the time of the alarm will not be updated correctly.
   The format of the alarm depends on, whether the error occurs in a code-procedure or in an external algol procedure, see the survey of alarm-messages on the following pages.

   An alarm is generally caused by the users program, for example if the procedures are called in a wrong order, or if the program does not care for unexpected values of result_cf.
   Some alarms may be due to an error in the file, as for example checksumerror in the filehead. A file-error may however be caused by a program-error in a previous run, or by combining files of different generations.
   A few of the errors should be quite impossible. They have the alarmtext -cf-error- and can only be due to some grave error in the cf-code, or to some hardware-error during the run.

Alarms from code-procedures.

All alarms from code-procedures have the following format:

```
<alarmtext> <alarmno>  cf-system
called from ...
```

where <alarmtext> is a short mnemonic cause, and <alarmno> a further specification.

The following survey of alarms is arranged alphabetically after the alarmtext.

text	alarmno	explanation	error caused by
array p	13	The parameter array is too short for the masterfile-key.	program
cf-error	10	The mother-record of the actual chain has disappeared.	cf-system
cf-error	37	The record-number inside a listfile-record does not correspond to the position of the record in the file.	cf-system
chain p	15	Parameter chainref does not contain a valid chainreference.	program
ch.ass.	9	The file and the chain-group are not associated.	program
ch.head	18	The head of a listfile-record is not consistent.	file
ch.state	16	The chain is not initialized, i.e. initchain has not been called after open-cf.	program
ch.state	17	Last accessed record is not defined, i.e. the chainstate has become empty after the last use of the chain.	program
ch.type	20	The chain is not headed, so a call of get-head is impossible.	program
d.state	29	The daughter-zone is in read-only-mode, so deletion of the mother-record and its daughter-chain is impossible.	program

express.	36	A return-parameter is given as an expression in the procedure-call.	program
mode p	11	Wrong mode-parameter in call of get-1, insert-1 or connect, i.e. mode<>1 and mode<>2 and mode<>3.	program
m.state	28	The mother-zone is in read-only-mode, so delete-1, insert-1 or connect in mode 1 (next to mother-record) is impossible.	program
no curr.	14	Current record in a listfile does not exist.	program
par.pair	<i>	An error in the parameter-list in the call of set-jumps-cf, get-param-cf or set-param-cf. If i > 0, i shows the number of the wrong parameterpair. i = 0 denotes an error in the parameter jumpproc in call of set-jumps-cf.	program
prep-cf	24	too few segments in the document of a listfile, i.e. segs < segs-in-head, or the number of segments is less than it was in the last run in update-mode.	file
prep-cf	25	The zonebuffer is too small to open a listfile.	program
prep-cf	26	Checksumerror or some other error in the filehead of a listfile.	file
prep-cf	32	The zone for a file is not declared with at least two shares.	program
prep-cf	33	Too many segments in the document of a listfile, i.e. (segs - segs-in-head)//segs-per-block > max-blocks.	file
prep i	1	Too few or too many segments in the document of a masterfile, i.e. segs < segs-per-buck or segs > segs-per-buck × max-bucks.	file
prep i	2	The filelength is less than it was in the last run in update-mode, or some error in the bucket-head.	file
prep i	3	The zonebuffer is too small to open or initialize a masterfile.	program

prep i	4	Checksumerror or some other error in the filehead of a masterfile.	file
prep i	5	The zone for a masterfile is not declared with three shares.	program
prep i	6	Wrong zonestate internally	cf-system
prep i	7	Empty masterfile	file
rec.no.	19	The record-number of a listfile-record is outside limits. This may happen explicitely in a call of get-numb-i as a program-error or implicitely in other procedures, if the file has been destroyed.	program or file
rec.no.	22	No listfile-record is assigned to the record-number. Program- or filererror as for alarmno. 19.	program or file
z.state	<i>	Wrong zonestate. <i> is the actual zonestate.	program

Alarms_from_external_algol_procedures.

Alarms from external algol procedures have the following format:

```
xxx<proc.name>              alarm:
<alarmtext> <integer>       ext <line-interval>
called from ...
```

An exception is alarms from the protection-system, which have the format:

```
xxxprotectcf                alarm:
file <fileno> <filename>    vers.in cat: <version>
<alarmtext> <integer>       ext <line-interval>
called from ...
```

Here the text -file- is replaced by the text -descr-, if the trouble concerns the description-file.

program	text	alarmno	explanation	error caused by
xxxbuflengthcf	block p	<I>	The parameter blocks-in-core has an illegal value. <I> is the erroneous value.	program
xxxbuflengthcf	prep-cf	0	Some error in the filehead.	file
xxxhead1	chains p	<I>	Chain-type or compressed-key-size in parameter array chains has an illegal value, or if I = 0 then wrong bounds of array chains, or if I > number of the last chain, then listfile not daughter of any chain group.	program
xxxhead1	loop-ch	0	A loop is found in the chain-structure given in parameter array chains.	program
xxxhead1	size-1 p	0	One of the values given in parameter array size-1 is illegal.	program
xxxheadm	chains p	<I>	As for procedure head-1	program
xxxheadm	recdescr	<I>	One of the values given in parameter array rec-descr is illegal, or if I > 2044 then too many keyfields. (Only for noofkeys > 50).	program
xxxheadm	head 1 p	0	Some unreasonable size parameter.	program
xxxheadm	head 1 p	1	Not room for 2 records of maxlength in one block.	program
xxxheadm	head 1 p	2	Not room for 1 block in the first bucket.	program

xxxnewreclcf	cf-error	<i>	Trouble with insert-m. <i> is the value of result-cf.	cf-system
xxxnewreclcf	fixed 1	0	The file contains records of fixed length, so it has no meaning to use new-recl-cf.	program
xxxnewreclcf	z.state	<i>	Wrong zonestate. <i> is the actual zonestate.	program
xxxprotectcf	change	<i>	The catalog entry with the name <filename> could not be changed. <i> is the result-value of the monitor-function.	job adm
xxxprotectcf	descrrec	<i>	The file-description-record in the description-file could not be fetched by get-m. <i> is the value of result-cf after get-m.	file
xxxprotectcf	lookup	<i>	The catalog entry with the name <filename> could not be looked up. <i> is the result-value of the monitor-function.	job adm
xxxprotectcf	reserve	<i>	The file with the name <filename> could not be reserved. <i> is the result-value of the monitor-function.	job adm
xxxprotectcf	setdescr	0	The procedure setdescr-cf was not called before open-cf.	program
xxxprotectcf	updmark	1	The file is in the state of an unterminated update.	file
xxxprotectcf	updmark	0	An updatemark was expected in the catalog entry of the file.	cf-system
xxxprotectcf	version	<i>	The version-number of the file does not correspond to the version-number in the description-file. <i> is the version-number in the description-file.	job adm

Survey of cf-procedures.

no.	procedure names and parameters	jmp	result-cf value	and meaning	current record
1×	buf_length_cf (filename, blocks_in_core)	-	1	ok	meaningless
2×	extend_cf (z, segments)	-	1	extended	unchanged
			2	ext, simple ins.	-
			>2	not extended monitor-error	-
3	open_cf (z, filename, giveup)	-	1	ok	zm:first; zl:none
			2	ok, simple insert	zm:first
4	set_jumps_cf (z, jump_proc, procno, results)	-	1	ok	unchanged
5	init_chain (z, zl, chainno, chainref)	+	1	ok	unchanged
6	close_cf (z, rel)	+	1	ok	none
7					
8	get_m (zm, key)	+	1	record found	the wanted
			2	not found	the next in file
			3	- - , eof	the first - -
9	get_l (zl, chainref, gmode)	+	1	record found	the wanted
			2	not found	gmode=2: last acc. else: none
10	get_head (zl, chainref, key)	+	1	ok	unchanged
			2	not connected	-
11	insert_m (zm, record)	+	1	inserted	the inserted
			2	already in file	the one in file
			3	too expensive	the next in file
			4	file full	- - - -
			5	length error	- - - -
			6	no buffer	- - - -
12	insert_l (zl, chainref, lcmode, record)	+	1	inserted	the inserted
			2	fill-limit exceeded	none
			3	length error	-
			4	no block	-

13	connect (z1, chainref1, chainref2, icmode)	+	1	connected	last acc.in chain1
			2	not connected (already conn.)	none
14	delete_m (zm)	+	1	deleted	the next in file
			2	- , eof	the first - -
			3	not del. last left	the one
15	delete_1 (z1, chainref)	+	1	deleted	the next in chain
			2	del, last in chain	none
16	delete_chain (z, chainref)	+	1	deleted	unchanged
			2	no chain to del.	-
17	next_m (zm)	+	1	found	the next in file
			2	not found, eof	the first - -
18	put_cf (z)	+	1	ok	unchanged
19	read_only_cf (z)	+	1	ok	unchanged
20	read_upd_cf (z)	+	1	ok	unchanged
21	update_all_cf (z)	+	1	ok	unchanged
22					
23	get_numb_1 (z1, recno)	+	1	record active	the wanted
			2	record dead	none
24×	new_rec1_cf (z, length)	-	1	changed	same, new length
			2	last rec. in file	- , old length
			3	too expensive	- , - -
			4	file full	- , - -
			5	length error	- , - -
			6	no buffer	- , - -
25×	head_m (filename, fileno, chains, recdescr, no_of_keys, size_m)	-	1	ok	meaningless
26×	head_1 (filename, fileno, chains, size_1)	-	1	ok	meaningless
27	init_file_m (zm, filename, giveup, buckfactor, blockfactor)	-	1	ok	none

28	init_rec_m (zm, record)	+	1	record added	none
			2	file full	-
			3	length error	-
			4	key error	-
29					
30	get_param_cf (z, paramno, val)	-	1	ok	unchanged
31	set_param_cf (z, paramno, val)	-	1	ok	unchanged
32×	set_descr_cf (descrfile)	-		unchanged	unchanged
33×	protect_cf (z, action)	-		unchanged	unchanged
34	init_extract (z, filename, giveup)	-	1	ok	none
35	init_add (z, filename, giveup, buckfactor, blockfactor)	-	1	ok	none
36	extract_cf (z, record)	-	1	ok	none
			2	end of file	-
37	add_cf (z, record)	-	1	record added	none
			2	file full	-
			3	length error	-
			4	key or recno.err.	-
			5	no block	-

Procedures marked with × are external algol procedures.
A + in the jmp-column means, that set-jumps-cf can be used upon this procedure.

A survey of cf-states.

Zonestates for masterfiles:

after-declaration	(value = 4). The zone has been declared, but not yet opened. This is also the state after a call of close-cf.
initialize-m	(value = 20). During initialization.
read-only-m	(value = 16). During processing of the file. Changes in records will not be reflected in the file. Updating procedures are illegal.
read-update-m	(value = 18). During processing of the file. A block of records is only transferred to the file, if an updating procedure has worked upon one of its records.
update-all-m	(value = 19). During processing of the file. All records will be transferred to the file.

Zonestates for listfiles:

after-declaration	(value = 4). As for masterfiles.
read-only-l	(value = 22). The analogy of read-only-m.
read-update-l	(value = 23). The analogy of read-update-m.
update-all-l	(value = 24). The analogy of update-all-l.

Zonestates for reorganization (masterfiles and listfiles):

extract-cf	(value = 17). During the extraction of records from a masterfile or a listfile. The state is set by procedure init-extract.
add-cf	(value = 21). During the addition of records to a masterfile or a listfile. The state is set by procedure init-add.

Chainstates:

not-init	The cf-procedure init-chain has not yes been called.
empty	There is not defined a last-accessed record for the chain.
last-accessed-def	There is defined a last-accessed record in the daughterfile of the chain.

Record-states of listfile-records:

active	The record can be processed via a chain or its recordnumber.
dead	The record has been deleted, but is still member of one ore more chains. (It cannot be processed).

Format_of_integer_array_chains.

The purpose of the array is to specify the connections between files in the cf-system, i.e. the chain groups.
Chains are represented by the identifications of the motherfile and the daughterfile and a chainnumber. The chainnumbers are indirectly given by the order in which the chainspecifications appear in the array chains, while the logical filenumbers, which identify the files in the system, must be supplied by the user. The user must take care that the filenumbers identify the files unambiguously. The array is used as parameter for the two procedures head_m and head_1.

Declaration_of_chains:
integer array chains(1:no_of_chains×4);

Chain_specification.
A chainspecification consists of 4 consecutive elements of the array and the first specification must start in element no. 1. The upper limit of the array will stop the specification. The 4 elements of a chainspecification should be initialized as follows:

1. mother_no, the file_no of the motherfile.
2. daughter_no, the file_no of the daughterfile.
3. chain_type, the value 1 denotes a headed chain, the value 0 a not headed chain.
4. compressed_key_size (equivalent to key_part_size, RCSL 55-D99 p. 4), the quantity gives the number of bytes occupied by a compressed key of a motherrecord. It may be calculated according to the following rules:

    1. if_motherfile_is_a_masterfile
       add 4 for each long- or real keyfield
       add 2 for each integer keyfield
       add 2 for two successive byte keyfields
       add 2 for each single byte keyfield
       (A field containing a length-specification is not counted).

    2. if_motherfile_is_a_listfile
       the size is 2.

The quantity is a return value of the procedures head_m and head_1 for all chaingroups of which the actual file is the mother, i.e. the user need not be troubled by the calculation, if he calls the head_ procedure of a motherfile before those of the corresponding daughterfiles.

Chain-numbers:
The chains are numbered by the natural numbers (1, 2, ...) according to their appearance in the array chains. The chain-number is a call value of the procedure init_chain.

This appendix defines the format of the description file as required by the protection system.

File format:
The file is a masterfile of variable length records.

Format of file description record:

field no	type	address	content
101	integer	2	record length >= 30
1	long	12	keyfield_1 = 2
2	long	16	keyfield_2 = filenumber
3	long	20	keyfield_3 = 0
5	integer	30	version_number >= 0, <= 8000000.

Comments:

field no	comment
1-3	the key consistes of field no 1 to 3, longs in ascending order.
1	this field is called the description type.
2	the file number is the number used as a parameter for head_m or head_1.
5	the version number is checked and updated by the protection system.

Format of a record extracted by procedure extract_cf:

field no	type	address	content
1	integer	2	total length of extr.record
2	integer	4	not used by the cf-procs
3	integer	6	record number
4	integer	8	user part size
5	array	8	user part
6	array	8+reclength	chain part

Comments:

field no	comment
1	total length = 8 + user_part_size + chain_part_size.
2	this field is intended for the checksum of invar/outvar.
3	the record number is the record number of a listfile record, and the natural number (1, 2, 3,...) of a master record.
4	the user part size is equal to the normal record length except in the case of variable length listfile records, where it is 4 bytes smaller because the first 4 bytes are not included in the user part.
5	the user part is a copy of that part of a record, which appears as a zone record, with the exception of the first 4 bytes of a variable length listfile record.
6	the size of the chain part is given in the description of the procedure head_1. A more detailed format of the chain part is given here: <chain part>::= <mother field> 0/n1 <daughter field> 0/n2 This notation means that a chain part consists of n1 mother fields followed by n2 daughter fields. n1 and n2 may be zero, but they are fixed for all records of a certain file. <mother field>::= integer (2 bytes) The mother fields are placed in chain group number order, one mother field corresponding to each chain group of which the file is the mother.

value of mother field:

0	end of chain, i.e. there are no daughter records.
> 0	the record number of the first daughter record.

<daughter field>::= <next field> <ref.to mother>0/1

The daughter fields are placed in chain group number order, one daughter field corresponding to each chain group of which the file is the daughter.

<next field>::= integer (2 bytes)

sign of <next field>

>= 0	the record is active.
< 0	the record is dead. The next fields of all daughter fields are negative in this case, and at least one daughter field will be not connected.

<next field> extract 23:

8388607 = all ones	the record is not connected to any chain of the chain group corresponding to this daughter field.
0	end of chain, i.e. this is the last record in this chain.
>0 and <8388607	the record number of the next daughter record in this chain.

<ref.to mother>::= <compressed key of mother>

This field is omitted if the chain group is not headed.
If the mother file is a listfile, this field is just a 2 byte integer holding the record number of the mother record.
If the mother file is a masterfile, the field is more complicated:
The keyfields of the mother record are laied out close to each other in the order of decreasing priority. (1 to no_of_keys).
The close packing is disturbed by byte keyfields, because a keyfield of type integer, long, or real must be preceeded by an even number of bytes in the compressed key.
A single byte keyfield between two keyfields of other type will thus require 2 bytes room in the compressed key, the first byte holding the keyfield, and the other being equal to zero.
Two succeeding byte keyfields are packed into 2 bytes.

Formulas for the zone bufferlength required by cf-files. The formulas will in some cases specify up to a few bytes more than actually needed.

Masterfiles:

```
required_bytes:=
  176
  + 8 × no_of_associated_chain_groups
  + 28 × no_of_keys
  + 8 × no_of_keyfields_of_type_long
  + if fixed_record_length then  0 else  12
  + (compressed_key_size + 4) × ((segments_in_file - 1)
     // segs_per_buck  +  3)
  + 512 × segs_per_block_table
  + 512 × segs_per_block × (if full_insert then  2 else  1);
```

The term involving compressed_key_size covers the room needed for the bucket table.
The quantity segs_per_block_table needs a specification. It is the number of segments occupied by the blocktable placed in the beginning of each bucket.

```
blocks_per_bucket:=
  512 × segs_per_buck // (512 × segs_per_block
  + compressed_key_size + 4);

segs_per_block_table:=
  ((compressed_key_size + 4) × blocks_per_bucket - 1)
  // 512 + 1;
```

Listfiles:

```
required_bytes:=
  84
  + 8 × no_of_chain_groups_of_which_file_is_mother
  + 18 × no_of_chain_groups_of_which_file_is_daughter
  + sum of (compressed_key_size of all mother files)
  + segments_in_file // segs_per_block // 4 × 2
  + (512 × segs_per_block + 2) × blocks_in_core;
```

```
CF-SYSTEM    Programming example.

begin
comment
  This is an example of an algol 6 program which creates 2
  master files: master_1 and master_2, and one listfile:
  list.
  2 chain groups: chain_1 and chain_2, are associated to
  master_1 and list, and to master_2 and list respective-
  ly.
  A rudimentary description file: descrfile, sufficient
  for the check of version numbers performed by the cf
  protection system is also created.
  Various functions are performed on the file configura-
  tion.
;


  procedure  check_one;
  comment    gives a case alarm if result_cf <> 1;
  case result_cf of begin end;


  procedure  printtime(text);
  string  text;
  comment
    prints the time consumed since last call;
  begin
    own boolean  later_call;
    own real  cpubase, timebase;
    real  cpu, time;

    if later_call then
    begin
      cpu:= systime(1, timebase, time) - cpubase;
      write(out, <:<10>:>, text, <: in seconds, cpu::>,
      <<dddd.dd>, cpu, <:, real::>, time);
    end  later_call
    else  later_call:= true;

    cpubase:= systime(1, 0, timebase);
  end  printtime;

  printtime(<::>);    blocks_read:= 0;

  begin
  comment
    block for creation of file heads;

    integer
      file_no,
      fixed_rec_length,
      i,
      max_blocks,
      max_bucks,
```

```
    max_rec_length,
    min_rec_length,
    no_of_keys,
    segs_per_block,
    segs_per_buck;

  integer array
    chains(1:(2×4)),
    rec_descr(1:4, 1:2),
    size_l, size_m(1:4);

comment
  initialize array chains:

  chain group     mother      daughter    chain type  compr.key
  1               1           100         headed      see head_m
  2               2           100         headed      see head_m
;

  for i:= 1 step 1 until 2×4 do
    chains(i):= case i of(
      1, 100,   1,   0,
      2, 100,   1,   0);

comment
  the fourth field in each line above, compressed keysize, is
  initialized by head_m, and used by head_l.
  (from the record description below it can be seen to be 8
    bytes).

  create the head of master_1;

  file_no:= 1;

comment
  initialize the record description:

  keyfield    type        order       address
  1           long        ascending    4
  2           byte        descending  11
  3           word        ascending   10
  length      fixed
;

  no_of_keys:= 3;
  for i:= 1 step 1 until (no_of_keys + 1) × 2 do
    rec_descr((i+1)//2, 2-i mod 2):= case i of(
      +3,   4,
      -1,  11,
      +2,  10,
       0,   0);
```

```
comment
  initialize size parameters;

  size_m(1):= max_rec_length:= 120;
  size_m(2):= max_bucks:=      100;
  size_m(3):= segs_per_buck:=   40;
  size_m(4):= segs_per_block:=   2;

comment
  create the file head, the backing store area: master1,
  must exist;

  head_m(<:master1:>, file_no, chains, rec_descr, no_of_keys,
         size_m);

comment
  for simplicity, the same parameters are used for master_2;

  file_no:= 2;

  head_m(<:master2:>, file_no, chains, rec_descr, no_of_keys,
         size_m);

comment
  create the description file head;

  file_no:= 1000;

comment
  initialize the record description according to appendix E:

  keyfield     type        order        address
  1            long        ascending    12
  2            long        ascending    16
  3            long        ascending    20
  length       word        -             2
;

  no_of_keys:= 3;
  for i:= 1 step 1 until (no_of_keys + 1) × 2 do
    rec_descr((i+1)//2, 2 - i mod 2):= case i of(
      +3,  12,
      +3,  16,
      +3,  20,
       2,   2);
```

```
comment
  initialize size_m, the description file is regarded as
  being a small file;

  size_m(1):= max_rec_length:= 100;
  size_m(2):= max_bucks:=       50;
  size_m(3):= segs_per_buck:=   10;
comment
  never choose a smaller value for segs_per_buck;
  size_m(4):= segs_per_block:=   1;

  head_m(<:descrfile:>, file_no, chains, rec_descr, no_of_keys,
         size_m);


comment
  create the listfile head:

  variable record length, minimum about 20 bytes;

  file_no:= 100;

  size_l(1):= fixed_rec_length:=  0;
  size_l(2):= min_rec_length:=   20;
  size_l(3):= segs_per_block:=    1;
  size_l(4):= max_blocks:=     2000;

  head_l(<:list:>, file_no, chains, size_l);

end  block for the creation of file heads;

printtime(<:file heads created  :>);
```

```
begin
comment
  block for initialization of master files.
  master_1, and master_2 are provided with a dummy record
  having all fields equal to zero, because open_cf requires
  that a master file contains at least one record.
  the description file is initialized with 4 file description
  records;

  zone
    zm1(buflength_cf(<:master1:>, 1), 3, stderror),
    zm2(buflength_cf(<:master2:>, 1), 3, stderror),
    zdescr(buflength_cf(<:descrfile:>, 1), 3, stderror);

  integer
    file_no;

  integer field
    descr_length;

  long field
    descr_key_1,
    descr_key_2,
    descr_key_3,
    l_fld;

  real array
    rec(1:50);

comment
  initialize the field variables for the description file;
  descr_ length:=  2;
  descr_key_1:=  12;
  descr_key_2:=  16;
  descr_key_3:=  20;

comment
  set all fields of array rec to zero;
  for l_fld:= 4 step 4 until 200 do  rec.l_fld:= 0;

comment
  initialize master_1 with one record having all fields
  equal to zero;

  init_file_m(zm1, <:master1:>, 0, 1, 1);
  init_rec_m(zm1, rec);
  checkone;
comment
  this procedure checks that result_cf was one, see the
  procedure declaration at the beginning of the program;

  close_cf(zm1, true);
```

```
comment
  the same is done for master_2;
  init_file_m(zm2, <:master2:>, 0, 1, 1);
  init_rec_m(zm2, rec);
  checkone;
  close_cf(zm2, true);

comment
  initialize the description file with 4 records, describing
  the files including the description file itself;

  init_file_m(zdescr, <:descrfile:>, 0, 1, 1);

  for file_no:= 1, 2, 100, 1000 do
  begin
  comment
    the file numbers of master_1, master_ 2, list, and
    descr_file;
    rec.descr_length:= 30;
    rec.descr_key_1:=   2;
    rec.descr_key_2:= file_no;
    rec.descr_key_3:=   0;

    init_rec_m(zdescr, rec);
    checkone;

  comment
    the version numbers are zero in the description records as
    well as in the catalog entries of the corresponding files,
    if the files were created by set in this way:
    master1= set 120, etc. just before the call of this
    program;
  end  for file_no;

  close_cf(zdescr, true);

comment
  the list file needs no initialization;
end  block for initialization;

printtime(<:files initialized   :>);
```

```
begin
comment
  block for processing of the file configuration:
  200 records are inserted in both master files, at random
  keys, and 1000 list records are connected to records
  in both files via chain group 1 and chain group 2;

  zone
    zm1(buflength_cf(<:master1:>, 2) + 10×12//4, 3, stderror),
    zm2(buflength_cf(<:master2:>, 2) + 10×12//4, 3, stderror),
    zl(buflength_cf(<:list:>, 3) + 100//8, 4, stderror);

comment
  the addition to buflength_cf provides for extra bufferlength
  for extensions of the files during the processing: 10 extra
  buckets for the master files, and 100 extra blocks for the
  listfile.

  the factor 12 in the expression for the master zone buffer
  length is equal to compressed_keysize + 4, see appendix G;

  integer
    i,
    ic_mode;

  integer field
    length,
    m_key_3;

  long field
    l_fld;

  real
    chain_ref_1,
    chain_ref_2;

  real array
    m_rec, l_rec(1:50);


  procedure  create_key;
  comment
    this procedure generates a pseudo random master key
    in array m_rec;
  begin
    own integer  ps_random;
    random(ps_random);
    m_rec.m_key_3:= ps_random mod 10000;
  end  create_key;
```

```
comment
  initialize the field variables;

  length:=   2;  comment  the length field of list records;
  m_key_3:= 10;  comment  see the file head creation;


  set_descr_cf(<:descrfile:>);

comment
  this call provides the cf-system with the name of the
  description file;

  open_cf(zm1, <:master1:>, 0);
  checkone;
  open_cf(zm2, <:master2:>, 0);
  checkone;
  open_cf(z1, <:list:>, 0);
comment
  the version numbers and the update marks have been checked,
  and the zone states are read_only;

  read_upd_cf(zm1);
  read_upd_cf(zm2);
  read_upd_cf(z1);
comment
  now the zone states are read_update, insertions are allowed,
  and the update marks are set in the catalog entries;

  init_chain(zm1, z1, 1, chain_ref_1);
  init_chain(zm2, z1, 2, chain_ref_2);
comment
  the 2 chain groups are ready for processing, the chain_refs
  are used to reference them;

  for l_fld:= 4 step 4 until 200 do
    m_rec.l_fld:= l_rec.l_fld:= 0;
```

```
      for i:= 1 step 1 until 200 do
      begin
      comment
        insert 200 master records in master_1, with random values
        of keyfield 3, and the other fields equal to zero;

make_a_key:
        create_key;

insert_m_rec:
        insert_m(zm1, m_rec);

        case result_cf of
        begin
        comment  1,  ok,  do nothing;
            ;
        comment  2,  record exists already, try another key;
            goto make_a_key;
        comment  3,  not inserted,  too expensive.
          this is not possible when param_cf has not been used
          to change the insertion parameters;
            checkone;
        comment  4,  the file is full,  extend the file with one
          bucket = 40 segments;
          begin
            extend_cf(zm1, 40);
            checkone;
            goto insert_m_rec;
          end  4;
        comment  5,  length error, not possible with fixed length;
            checkone;
        comment  6,  no buffer, not possible because result_cf has
          been checked after open_cf and extend_cf;
            checkone
        end  case result_cf;

      end  insertion of 200 records in master_1;
```

```
comment
  insert 200 records in master_2 in a more crude way;

  for i:= 1 step 1 until 200 do
  begin
    create_key;

    insert_m(zm2, m_rec);
    case  result_cf of
    begin
    comment  1,  ok;
        ;
    comment  2,  exists already, repeat;
        i:= i - 1
    end  case result_cf;
  comment
    other results will give a case alarm;
  end  insertion of 200 records in master_2;

  printtime(<:master recs inserted:>);


  for i:= 1 step 1 until 1000 do
  begin
  comment
    insert 1000 list records connected to random master
    records.
    the list records are clustered in chain group 1, i.e.,
    insert_1 works upon chain_ref_1;

    create_key;
    get_m(zm1, m_rec);
  comment
    the result is ignored, there will always be a current
    record in a master file;

  comment
    insert a list record as the last in the chain_1 depar-
    ting from the current master_1 record.
    insertion as the first in chain is faster, but
    it does not demonstrate the use of get_1;

    get_1(z1, chain_ref_1, 1);
  comment
    read the first record in this chain, if any;

    ic_mode:= if result_cf = 1 then  2 else  1;
  comment
    insert mode is next to last accessed, if there is any
    record in the chain, else next to mother;
```

```
      for i:= 1 while result_cf = 1 do  get_1(z1, chain_ref_1, 2);
    comment
      read all records in the chain, last accessed in chain
      group 1 is now the last in chain, if any;

      1_rec.length:= 30;

insert_1_rec:
      insert_1(z1, chain_ref_1, ic_mode, 1_rec);

      case result_cf of
      begin
      comment  1,  ok, do nothing;
          ;
      comment  2,  fill limit exceeded, extend the file with
        20 blocks = 20 segments;
        begin
extend_the_file:
          extend_cf(z1, 20);
          checkone;
          goto insert_1_rec;
        end  2;
      comment  3,  length error;
        checkone;
      comment  4,  no block can take this record;
        goto extend_the_file
      end  case result_cf;

    comment
      connect the list record to a random master_2 record, as
      first in chain;

      create_key;
      get_m(zm2, m_rec);

      ic_mode:= 1;  comment  connect next to mother;

      connect(z1, chain_ref_1, chain_ref_2, ic_mode);
      checkone;
    end  insert 1000 list records;

  comment
    master_1 is not updated any more;
    read_only_cf(zm1);

    printtime(<:list recs inserted  :>);
```

```
comment
  go through all chains of chain group 2, at the same time
  look up the master_1 record being the mother of the chain
  1 passing through each list record, and at last delete the
  list record.
  the list records are counted, to check that all 1000 have
  been deleted;

comment
  master_2 is read by means of next_m, starting at the dummy
  record created by init_rec_m;

  m_rec.m_key_3:= 0;
  get_m(zm2, m_rec);
  checkone;

  i:= 0;
  for i:= i while result_cf = 1 do
  begin
  comment
    read the first record in the chain_1 departing from the
    current record of master_2;

    get_1(z1, chain_ref_2, 1);

    for i:= i while result_cf = 1 do
    begin
      get_head(z1, chain_ref_1, m_rec);
      checkone;
    comment
      now m_rec contains the key of the record, which is the
      mother of the chain_1 passing through the current list
      record;

      get_m(zm1, m_rec);
      checkone;
    comment
      the calls of get_head and get_m above are performed
      as a demonstration of how each list record acts as a
      link between a record in master_2 and a record in mas-
      ter_1;

      delete_1(z1, chain_ref_2);
      i:= i + 1;
    comment
      delete and count the list file record, delete will
      access the next record in chain_2, if any;
    end  reading and deleting of one chain;

    next_m(zm2);
  comment
    read the next master_2 record;
  end  reading of master_2;
```

```
    if i <> 1000 then
      write(out, <:<10>xxxerror in count    :>, i);

    close_cf(zm1, true);
    close_cf(zm2, true);
    close_cf(z1, true);
  end  block for processing of file configuration;

  printtime(<:list records deleted:>);

  write(out, <:<10>blocks read:  :>, blocks_read);
end  program
```

A run of the programming example.

The files were dimensioned to be filled up to about 70 percent.

Master_1 and _2 were situated on disc_1 and the listfile on disc_2 (see lookup cat.yes in the output).

The disc stores were of type RC 433.

The cpu and the disc stores were slightly loaded by other processes.

Note that the version numbers in the catalog entries of the 3 files have been increased to 1 during the run.

Output from the run.

```
×master1=set 80
×master2=set 80
×list=set 140
×descrfile=set 10
×cfex

file heads created   in seconds, cpu:    0.82, real:    2.47
files initialized    in seconds, cpu:    0.16, real:    1.32
master recs inserted in seconds, cpu:    9.23, real:  214.16
list recs inserted   in seconds, cpu:   26.26, real:  510.76
list records deleted in seconds, cpu:   10.22, real:  178.85
blocks read:   103
end
×lookup cat.yes master1 master2 list descrfile

master1 15 0 27 1634
  80 0 1 0 0 0 0
master2 17 0 27 3842
  80 0 1 0 0 0 0
list 3 0 27 3.872
  140 0 1 0 0 0 0
descrfile 22 0 27 820
  10 0 0 0 0 0 0
```

Appendix J. How to dimension the files.

This appendix contains some rules for the choice of the size parameters for the two procedures head_m and head_1.
The rules are based on one years experience with file configurations for administrative data processing.

The size_m parameters of head_m.

The 4 parameters are described in the order of occurrence in array size_m, a more natural order of specification is: max_rec_length, segs_per_block, segs_per_buck and max_bucks.

max_rec_length

The maximum length in bytes of the user part of a record.

The sum of max_rec_length and the size of the chain part must not exceed 512 × segs_per_block//2, i.e. half the block size.
The size of the chain part is 2 × number_of-_associated_chain_groups, see appendix F., format of extracted records.

Note that both max_rec_length, chain_part_size, and the actual record lengths are rounded up to a multiplum of 4 in case of variable record length.
In the case of fixed record length, max_rec_length and chain_part_size are rounded if the sum max_rec_length + chain_part_size is not a multiplum of 4.

In case of variable record length the value of max_rec_length should not be specified much greater than the actual maximum record length, because that tends to decrease the efficiency of insertions.

max_bucks

The maximum number of buckets the file will ever hold.

This quantity should be chosen high (f.ex. 8000 //segs_per_buck = max_bucks for a whole RC 433 disc store). The only cost is max_bucks × (compressed_key_size + 4) bytes of backing storage for the bucket table. (Normally only a few segments in the head of the file).
The amount of core store used for the bucket table in the zone buffer depends only on the actual size of the file.
For compressed_key_size see appendix D., format of array chains.

segs_per_buck

The number of segments in one bucket.

The quantity segs_per_block should be selected before segs_per_buck.

Segs_per_buck should not be chosen too small, especially not so small that only one block is left in the first bucket, because this will disturb the insertion of new records seriously.

A magic number concerning segs_per_buck is 40, the number of segments of one cylinder of the RC 433 disc store.
With each bucket equal to a cylinder of the disc store, the maximum number of cylinder shifts required for a call of get_m is one, against two in the general case.
On the other hand it is not quite simple to synchronize buckets and cylinders in practice.

In the following segs_per_buck is selected as to economize the use of core storage and backing storage for bucket table and block tables.
The block table always needs an integral number of segments both in the file and in the zone buffer, whereas the bucket table in the zone buffer just demands room corresponding to the actual number of buckets.
This suggests a bucket size which is so great that the entries in the block table utilizes an area which is just below or equal to an integral number of segments.
If the size of the block table is called segs_per_block_table, then segs_per_buck can be calculated thus:

```
segs_per_buck =
  (segs_per_block_table × 512//
   (compressed_key_size + 4))
  × segs_per_block + segs_per_block_table
```

The compressed_key_size is the total size in bytes of all keyfields of a record, see appendix D., format of array chains.

Normally segs_per_block_table can be set to 1, but in case of a great value of compressed_key_size or if the file is very great this may give rise to too small buckets and a bucket table of excessive size.
Balance between bucket table and block table is achieved if the value of segs_per_buck is not far from:

```
square_root(max_segs_in_file × segs_per_block)
```

I.e. the mean proportional of the file size and the block size.

But, segs_per_buck should not be selected too small, as a small bucket size will decrease the insertion efficiency, and it should in any case not be less than the value which makes the first bucket contain 2 blocks:

```
segs_per_buck >=
  3
  + ((compressed_key_size + 4) × max_bucks
     + 9)//512
  + segs_per_block_table
  + 2 × segs_per_block
```

If the value of segs_per_buck is not set below 40 segments this problem is unlikely to occur, and on the other hand there is no reason in normal cases to go below the 40 segments.

segs_per_block

The number of segments in one block.

A reasonable number of records should fit into one block, say 5 or more. This minimizes the loss of backing storage and increases the speed of a sequential reading.
On the other hand room is reserved in core for up to 2 blocks during the processing, so in case of great record lengths it might be better to use a shorter blocklength.

The balance between the core store demands of bucket table, block table, and block should also be taken into consideration, especially in connection with greater files.
The two aspects are included in the following formula:

```
segs_per_block = maximum_of
  (5 × (max_rec_length + chain_part_size) // 512 + 1)
and
  cube_root(max_segs_in_file ×
            ((compressed_key_size + 4)/256)××2))
```

The first expression will let a block contain a

reasonable number of records.

The second one will let the block table and the bucket table together use about as much room as one block, if the value of segs_per_buck is selected according to the rules in this appendix.

The quantity max_segs_in_file can be estimated as the the maximum volume of records plus 20 to 30 percent extra for administrative tables and spare room.

Example of a great master file.

```
max_rec_length = 150 bytes
chain_part_size = 10 bytes (5 chain groups)
compressed_key_size = 8 bytes (2 long keyfields)
max_segs_in_file = 8000 segments (one RC 433)
```

The first quantity to calculate is segs_per_block:

```
segs_per_block = maximum_of
  (5 × (150 + 10) // 512 + 1) = 2
and
  cube_root(8000 × ((8 + 4)/256)××2)) =
  cube_root(17.6) = 3 (the rounded value)
```

The last expression is decisive, we choose: segs_per_block = 3.

The next quantity is segs_per_buck. For segs_per_block_table equal to 1 and 2 we get respectively:

```
segs_per_buck = (1 × 512 // (8 + 4)) × 3 + 1
              = 127
and
segs_per_buck = (2 × 512 // (8 + 4)) × 3 + 2
              = 257
```

These values are compared with the expression:

```
square_root(max_segs_in_file × segs_per_block)
= square_root(8000 × 3) = 155
```

The choice of segs_per_block_table = 1 gives the best fitting to this value, so the conclusion is: segs_per_buck = 127.

Max_bucks is just set to 8000//127 = 63.

The size_1 parameters of head_1.

The 4 parameters are described in the order of occurrence in array size_1, which is also a reasonable order of specification.

fixed_rec_length

The fixed record length if the value is positive. If it is zero, variable record length is specified.

It is emphasized that fixed record length gives advantages concerning reorganization.
The value of this parameter depends entirely on the format of the users records. Fixed_rec_length is rounded up to a multiplum of 2, not 4 (see max_rec_length for master files).

min_rec_length

In case of variable record length this parameter specifies the minimum length of records which should be able to fill a block entirely.

It should not be chosen too great because it can be necessary to increase its value in connection with reorganization.
The waist of backing storage depending on min_rec_length is given by this formula:

100/(min_rec_length + chain_part_size + 1) percent.

segs_per_block

The number of segments in a block.

The block length should be so great that a reasonable number of records can go into one block. This number should not be less than 5 and not less than the average number of records in a clustered chain.

It is also of importance that each block demands half a byte of core store for a block table entry, i.e. a file of 1000 blocks demands about one segment of core for the block table.
If the block table shall not take up more room than half a block the following formula arises:

```
segs_per_block = maximum_of
  ((5 or number_of_recs_in_clustered_chain)
   × (max_record_length + chain_part_size)
   //512 + 1)
and
  square_root(max_segs_in_file/512)
```

For chain_part_size see appendix F., format of extracted records.

This means, (the last term), that a file of more than 500 segments should have segs_per_block >= 2, and that a file of more than 2000 segments should have segs_per_block >= 3.

max_blocks

The maximum number of blocks the file will ever hold.

This quantity should be chosen high (f.ex. 8000 //segs_per_block = max_blocks for a whole RC 433 disc store). The cost is only max_blocks//2 bytes of backing storage in the block table. (Normally only a few segments in the head of the file).
The amount of core store used for the block table in the zone buffer depends only on the actual size of the file.

A_list_of_keywords_for_the_cf-sysyem.

alarm	Unintelligent use of the cf-system will terminate the run with an algol runtime alarm. The alarm is identified by a short alarmtext, see the survey of these in app. A.
associated chains	A term used in procedure descriptions for the chaingroups, that are defined for a specific file.
buckets	See RCSL 55-D99, file-i.
cf_proc_no	An integer call value to the users jump procedure giving the proc.no. of the procedure last called. Every cf-procedure has a procedure number which may may be found in the head line of the procedure description. The number is also used in calls of set_jumps_cf to specify when the jump procedure should be called.
chain	A term for listrecords with common head.
chainfield	A field in the protected part of a record used for linking. The format of the field depends on the type of the chain.
chaingroup	All the chains connecting two specific files by means of one set of chain fields.
chainno	A number of a chain-group. (See the procedure description of init_chain, and the description of array chains, app. D).
chainref	The reference for a specific connection between two files. This reference is created by the procedure init_chain, and is used as parameter in several procedures. (See the procedure descriptions).
chainstate	Some of the cf-procedures are dependent on the latest use of a specified chain. The chainstate keeps track of that. See the possibilities in the survey of the cf-states, app. C.
current record	A term for the last processed record in a file. Current record is the same as the zonerecord.
daughterfile	The subordinate file of a chain, i.e. the file that contains the elements of a chain. (always a listfile).
filename	The name of a backing store area.
file_no	The logical number of a file used in chain

specifications. (See the description of array chains, app. D).

head — A term for a record in a motherfile containing the record number of the first record in a chain. Chains are said to be headed if all records in a chain contain the reference to the head.

key — A group of fields in a masterfile record used for identification and organization. When used as a parameter of a procedure, a real array with the same format as a record (see record) long enough to hold all keyfields.

jump_proc — An exit procedure specified by the user. See app. A (Errors during processing, -Unnormal situations), and the procedure description of set_jumps_cf.

last_accessed record — The record number of a daughterfile record, which has been last accessed via a specific chain-group.

listfile — Is either a daughterfile or a daughterfile and a motherfile. Records are referred to by a record-number (see recordno.). Characteristics of listfiles are that they are badly accessed sequentially, and that insertion of records is done according to a strategy, so that the user cannot determine the physical address or record number of the new record.

list_record_state — Every record in a listfile has an indication of its -state-. See the possibilities in the survey of the cf-states, app. C.

masterfile — Is always a motherfile. Records are referred to and identified by a key, and the organization is indexed sequential. See RCSL 55-D99.

max_rec_length — Is for a masterfile less than segs_per_block × 256. For a listfile see the calculation in procedure description for procedure head_1.

min_rec_length — Is the length of a record, which can hold the whole key and lengthfield.

motherfile — A term for a file that contains the head-records of a chain-group. May be a masterfile or a listfile.

originating in — I.e. rooted in. A term used only in the procedure descriptions of the delete-procedures.

proc.no. — See cf_proc_no.

result_cf — A standard integer variable used to designate the result of a call of a cf-procedure.

rec_no_cf — A standard integer variable holding the last delivered rec_no in listfiles.

record — A number of consecutive bytes. When used as a parameter of a procedure the elements must be stored in the lexicographical first elements of an arbitrary array. The record may hold a length specification.

recordlength — A cf-file may consist of either variable length or fixed length records. If fixed length is chosen, all records are of max_rec_length. Recordlength is always given as the number of bytes of the users part of the record.

recordno. — (short: rec-no). Records in listfiles are identified and referred to by record numbers, which are allocated by the cf-system during the insertion.

zonestate — The cf-procedures are dependent on the latest use of the zone. The zonestate keeps track of that. See the possible zonestates in the survey of cf-states, app. C.