limes datentechnik

FLAM

FRANKENSTEIN-LIMES-ACCESS-METHOD

for UNIX

USER MANUAL

— October 2014 Version 4.3 —

Phone ++49 6172 / 5919-0 ■ Fax ++49 6172 / 5919-39
http://www.flam.de

User Manual FLAM V4.3 (UNIX)

FLAM (UNIX)

User Manual

Contents

Contents

FLAM (UNIX)

User Manual

Chapter 1:

FLAM Overview

1. FLAM Overview

FLAM is an implementation of the FLAM algorithms for computers running under the UNIX operating system. Its purpose is to compress and decompress files in a format that allows easy exchange of data with numerous computer types from a wide range of manufacturers. Moreover, it offers the option of encrypting compressed or uncompressed data.

Its heterogeneous compatibility also renders FLAM particularly suitable for long-term archiving of data, since the reproducibility of this data is no longer tied to the life cycle of a specific system. In addition, FLAM enables files to be converted in almost any way, i.e. to files with a different organization, a different record format or a different character set.

The architecture of the software is layer-oriented, i.e. it is made up of a series of hierarchically arranged functional groups which are completely distinct from one another and which communicate via defined interfaces. Since these interfaces are open, it is up to the user to choose the extent to which he wishes FLAM to be integrated in his applications.

FLAM can be invoked either by means of a command or by an application program via the corresponding interface. Although FLAM is capable of compressing any type of file, it was originally designed for files with a largely standardized record structure and character-coded data. It is most efficient at compressing such files. The above requirements are normally satisfied if the processing application programs are written in a high-level programming language. This chapter attempts to assist the user in deciding when, and if so how, FLAM can be used effectively and with the optimum benefit.

The flam command may contain the necessary information either in the form of parameters in the command line itself or as FLAM parameters in a parameter file, which is referred to by the parameter parfile=parameter file. Chapter 2, "The flam command", discusses these parameters in greater detail. Unless otherwise explicitly mentioned, all the information in this chapter relating to the use of parameters in the command line applies equally to the parameters in the parameter file.

1.1 Mode of Operation

When a file is being compressed with FLAM, one or more original files are read record by record, and a compressed (and optionally encrypted) file, called FLAMFILE, is created. When a file is decompressed using FLAM, the compressed file is read sequentially and one or more decompressed files are created.

The compression process consists of a series of recurring cycles. Each cycle generally entails reading a fixed number of records in the original file; these records are then buffered in the matrix buffer, compressed and output in the form of a compressed block. The number of records is between 1 and 255 or 4095, depending on the compression mode used, and is specified when the compression procedure is invoked. A compressed block containing correspondingly fewer records is generated when the end of the file is reached or if the matrix buffer overflows.

The compressed blocks created by FLAM are written into a sequential FLAMFILE.

By the record-oriented reading, the file's structural information is preserved, which is of great importance for reproducing the original file in computing environments with data management systems.

Depending on the compression mode, each record in the FLAMFILE is given either a binary checksum or a check character, which is used to verify the integrity of the data when the file is decompressed. The FLAMFILE can be saved or exchanged with another computer. When the file is decompressed, it may have a different organization, a different record format or a different record size from the original file, depending on the settings entered by the user or on the default settings.

1.2 FLAM Modes

FLAM supports four compression modes: adc, cx7, cx8, and vr8. In the adc, cx8, and vr8 modes, it creates the FLAMFILE as a binary file. Mode adc is the most efficient one. It compresses the datastream by removing the redundancy of repetitive sequences and can be used universally. The other modes mainly eliminate "vertical redundancy", i.e. their efficiency relies on the recognition of re-occurring column contents in table-structured data.

Compression mode adc can be combined with one of two encryption methods supported by FLAM. The first is AES, that has been declared standard for US government agencies by NIST (National Institute of Standards and Technology), the second is a proprietary algoriths of limes datentechnik gmbh. The FLAM modes **aes** and **flamenc**,

respectively, correspond to these two combinations. None of the other compression methods allows encryption.

In all binary modes the compressed information is largely independent of the characteristics of the system on which the compression took place, i.e. the FLAMFILEs must not be modified, for example, if they are transmitted on a line which translates codes automatically.

FLAMFILEs which are compressed in the cx7 mode, on the other hand, are insensitive to code translations, since they represent all control information by means of alphanumeric characters, the codes for which are standardized internationally in the ASCII and EBCDIC systems. Providing the original data is character-coded and suitable for exchange between systems with different character sets,

the information content of the FLAMFILE remains fully intact after it has been converted to the character set of the destination system, despite the altered binary contents, thanks to the character-oriented interpretation.

The compression mode is identified automatically when the file is decompressed. The user thus does not need to know the compression mode in order to decompress a file.

1.3 Matrix Buffer and Number of Records

The size of the matrix buffer and the number of records which are stored there can be specified for the compression procedure using the maxbuffer and maxrecords parameters. As a general rule, the efficiency of the compression procedure increases the more records are compressed together in the same block. If there are no other factors which need to be taken into account, it is normally best to choose the highest possible number (255 or 4095, resp.). This preset number is, however, merely an upper limit. There may be fewer records if the space in the matrix buffer is exhausted or if the end of the original file is reached.

The size which must be set for the matrix buffer is dependent on the number and size of the records that need to be buffered.The space requirement for files with fixed-size records is calculated simply as the product of the number of records and the record size. If the files have a variable record size, it is not normally possible to predict the memory requirement exactly. The value which is chosen is bound to be approximate. However, the matrix buffer must always be at least large enough to hold every individual record in the file.

1.4 The FLAMFILE

1.4.1 Structure of the FLAMFILE

In order to achieve the aim of universal interchangeability, the structure of the FLAMFILE must be such that it can be represented in any operating system. This "least common denominator" is the sequential file with fixed-size records. Although other FLAMFILE record formats are allowed for the FLAMFILE in addition to the fixed format, the compressed data which is generated by the compression procedure is broken down into "abstract" records - known as FLAM records - with an identical size; these records are then mapped to the "concrete" structures of the installed operating system. In addition to the compressed data, each FLAM record contains a FLAM record size field at the beginning and either a checksum or a check character at the end. The FLAM record size field and the checksum or the check character are elements of the FLAM syntax. This permits the start and end of a FLAM record to be identified irrespective of the record format in which the FLAMFILE has been stored by the operating system.

In FLAM terminology, a FLAM record is the same as a record if the FLAMFILE has fixed and variable record formats. The specified FLAM record size is thus the same as the record size or the maximum record size.

The residual data in a compressed block belonging to a FLAMFILE is merged with the data in the next block to form a FLAM record with the fixed or maximum size. With the possible exception of the final record, all the FLAM records have the same length. This enables optimum use to be made of the storage medium; however, it is not possible to assign the records of the original file to the records of a FLAMFILE uniquely. A FLAMFILE of this type can therefore only ever be decompressed in its entirety.

FLAM supports fixed and variable or stream record formats for the FLAMFILE. Irrespective of the record format, the minimum permissible size of a FLAM record is 80 bytes and the maximum size 32,760 bytes (4095 for mode=cx7), providing a lower upper limit has not been set by the user. This span is sufficient in practice to allow any FLAMFILE to be transferred, even in situations where this would not necessarily be the case with the original file, owing to restrictions imposed on the record size or the record format by the file transfer product.

Even though many data exchange problems can be eliminated to a greater or lesser extent with the FLAM concept, there are some which still cannot be solved. If a file is unsuitable for uncompressed transfer with a particular file transfer product, for example on account of the binary data it contains, it will still not be possible to transfer it after it has been compressed with FLAM, even if it is compressed in the cx7 mode. The reason for this is that every bit combination which occurs in the original file also occurs in the FLAMFILE.

1.4.2 The FLAM File Header

In addition to the compressed data, a FLAMFILE may contain other information which is stored in the FLAM file header. FLAM distinguishes between two information categories in two different types of file header:

Common information is stored in the common FLAM file header. This includes all information about the original file, such as the file organization, record format, length and position of the primary key, etc., as well as details of the FLAM version and the operating system in which the FLAMFILE was created.

User-specific information of any kind is stored in the user-specific FLAM file header. This header type can be inserted additionally if the record interface is used, for example for information which must be evaluated by the application program.

Creation of the common FLAM file header is supported by all interfaces (command, subprogram and record) by means of parameters and arguments. A user-specific FLAM file header can only be created if the record interface is used. This interface incorporates the following functions for creating and evaluating the various file header types:

FLAM file header type	Created by	Evaluated by
Common	flmphd	flmghd
User-specific	flmpuh	flmguh

It is not mandatory to insert FLAM file headers in the FLAMFILE. If they are inserted, it is essential to observe the stipulated order when invoking the functions of the record interface (see section 1.5.3 for further details).

FLAM file headers are always inserted in the FLAMFILE directly preceding the compressed version of the file to which they belong. If a FLAMFILE contains compressed versions of several files, a FLAM file header can be inserted in front of each compressed file.

When the files are decompressed, the compressed data originating from different files can be distinguished and decompressed into separate files.

1.4.3 Secure FLAMFILEs

Each record in a FLAMFILE is protected against manipulations and transmission errors by an appended checksum. In compliance with increased requirements regarding cryptographic security, additional security features for encrypted FLAMFILEs have been implemented which improve the protection against a greater variety of attacks. A FLAMFILE provided with these features is called "Secure FLAMFILE".

The new protection mechanisms work on three levels
- the segment level,
- the member level, and
- the file level.

Here, "segment" denotes the compressed result of a single matrix and "member" the entirety of segments from a single original file.

In a Secure FLAMFILE, there are byte and record counts both for each member as well as for the entire file, members are numbered consecutively, and a timestamp marks the creation time. When encrypted with AES, MACs (Message Authentication Codes) are attached to the FLAMFILE. Those are checksums calculated with a secret key that allow verifying the authenticity of the data. A MAC is calculated for each segment. To ensure the completeness of a member, a second MAC is chained through all segments of the same member, and a third MAC, chained through all members of the FLAMFILE, ensures the completeness with regard to the file. Similar features are included when FLAMFILEs are created with mode=flamenc.

To include the information necessary for these checks, such as counts, timestamps, MACs, etc., a member header is inserted in front of each member and each member ist followed by a member trailer. In addition, a file trailer is appended at the end of every Secure FLAMFILE.

Secure FLAMFILEs may not be concatinated since this would create discontinuities of numbering, counters, and MAC chainings that would make FLAM assume integity violations and break off processing. Individual members of a FLAMFILE, however, can be extracted, since checking of the member chaining is suspended, whenever members are skipped, and only the segment chaining is then tested.

1.5 Interfaces

FLAM's various interfaces are extremely flexible.

FLAM can be embedded in user-defined applications or vice versa by means of calls at different levels. The available interfaces are listed in the table below:

Interface	Call level	Purpose
FLAM command	System	Compress/decompress entire files interactively or in procedures
flamup subprogram	Application programs	Compress/decompress entire files in user-defined applications
Flamrec record interface	Application programs	Transfer and recall individual records in user-defined appliations
User-defined input/output	FLAM	Replace FLAM input/output routines
User exits	FLAM	User-defined file and record preprocessing and postprocessing for compressed and non-compressed data Key management

These interfaces have a hierarchical order, which defines which interface can be used or invoked from where. The interfaces are subdivided into two categories: active and passive.

Active interface calls are initiated either by the user or by his application program; they activate the FLAM routines which support them. These routines include the command and the subprogram and record interfaces. FLAM uses passive interfaces to activate user routines which are required to perform a particular range of functions, but whose calls are not synchronized with the logic of the application program. This category includes the user-defined input/output and the user exits.

The next few sections contain a brief description of the ways in which the different interfaces can be used, without covering every single detail. Each of the following chapters contains a detailed description of one of the interfaces and can be used as a reference document. All the program interfaces, i.e. all interfaces other than the command, are represented in C notation in the reference chapters, and the C header file flamincl.h containing the C constant definitions is provided for C programmers.

However, these interfaces do not support C programs only. Rather, they can be used with any programming language, since all arguments are referenced by addresses instead of the typical C referencing by values.

Even though, in a strict C context, a name without an "*" operator still identifies the argument address with this form of transfer, all references to such arguments in the descriptions below in fact relate to the argument values.

1.5.1 The flam Command

The flam command can be used to compress or decompress individual files or groups of files either interactively or by means of procedures. The compress or decompress parameter specifies which operation is to be performed. All the information necessary to execute the command can be specified in the form of inputs made directly in the command line, in a parameter file or as installation-specific default values.

If the flam command is specified with the list parameter, a list of the currently active settings of the default values is displayed; they can be modified by the system manager using the -defaults parameter.

The flam command permits a separate FLAMFILE to be created from each original file. It is also possible to compress a group of original files into one FLAMFILE.

The input files which are compressed can either be named explicitly or specified implicitly by means of patterns with wildcards; either explicit names or names determined through substitution rules are allowed for the output files.

The same specification forms can also be used when the files are decompressed; in addition, the special output specification [ ] or [*] permits a file to be restored with its original name and attributes.

The attributes of the decompressed file, such as its organization and record format, may otherwise differ from those of the original file.

FLAM can thus also be used for file conversions.

With the exception of the -list and -defaults parameters, there are equivalent FLAM parameters for all the elements of the command syntax, so that each setting can be entered either directly in the command line or in a parameter file which is referred to in the command by the parameter parfile=*parameter file*. Using a parameter file permits complicated keyboard inputs to be avoided, for example if settings which differ from the installed default values are required frequently.

1.5.2 The flamup Subprogram

The interface to the flamup subprogram permits roughly the same operations as with the flam command to be initiated in an application program.

This interface makes the same range of functions available to the application programmer as to the user, with the exception of the functions for listing and modifying default settings. FLAM parameters are used to specify these settings, similar to the procedure for the parameter file. These parameters are specified consecutively in an ASCII string, which is transferred to the subprogram in the form of an argument. Once again, it is possible to refer to a parameter file using the parameter parfile=*parameter file*. If any information is omitted, the same default settings are used as for the flam command.

When flamup has been executed in its entirety, the invoking program receives a return code indicating either the success of the call or the cause of the error.

1.5.3 The flamrec Record Interface

The flamrec record interface permits file processing and compression to be integrated in an application program. Decompressed records can be read sequentially from a FLAMFILE. A new FLAMFILE is created when the compressed data is output. A FLAMFILE can thus be opened either for read accesses only (decompression) or for write accesses only (compression).

FLAM's record interface consists of a class of functions, which allow the application program to access and process compressed data in the same way as is possible with ordinary input/output instructions in the case of non-compressed data. The use of these functions is governed by rules similar to those which apply to the standard input/output. After the file has been opened, records can be read or written; the file is closed again when all processing has been completed.

The compression or decompression that takes place during processing is completely transparent to the programmer, so that compared to programs using conventional input/output operations, there are almost no differences in the program logic.

By using the record interface, compressing (or decompressing) the processed data no longer requires an extra step after (or before) running the application. In addition, it is no longer necessary to provide sufficient temporary storage space for both the compressed and the uncompressed copy of a file.

1.5.4 The User-Defined Input/Output

FLAM supports all inputs and outputs to and from disk files using standard input/output instructions. However, it also allows the user to replace these standard input/output routines with routines of his own. This may be a good idea, for example, if there is a possibility of an input/output device other than a hard disk being used or if the file contains a record format which is not supported.

It is not possible to describe every single aspect of every function of the user-defined input/output in detail, since these vary according to the device and to the nature of the data. Data which is received via a modem line, for example, requires different actions from data supplied by measuring instruments. FLAM considers the origin of the data to be "file", even though in a particular instance other objects may be involved.

The functions of the user-defined input/output are as follows:

Function	Purpose	Corresponding I-O calls
usrcls	Terminating processing of file	fclose
usrget	Reading one record	fgets
usropn	Opening file	fopen
usrpos	Positioning to record	fseek
usrput	Writing one record	fputs

FLAM specifies a range of functions based on the file processing requirements for this interface, in accordance with the above point of view. It is up to the user of the interface to optimize the design of the functions, to ensure that the desired results are achieved with them.

In order to be able to use a user-defined input/output, the routines concerned must be linked to the application program. This option is therefore only supported by the program interfaces, i.e. by the subprogram and record interfaces.

The support by the record interface of course only applies to accesses to the FLAMFILE, since these are the only types of access which take place via it.

A user-defined input/output is initiated by invoking flmopd with device=7. This input/output can be activated at the subprogram interface for all the files which are concerned by the compression or decompression procedure, by means of the FLAM parameters user_io, inuser_io and outuser_io.

Whereas, with the functions of the record interface, the programmer is obliged to observe a series of rules regarding the order and consistency of the calls, the correct interaction of the functions of the user-defined input/output is the responsibility of FLAM, which also takes the initiative for the calls.

The programmer of the user-defined input/output routines is merely required to make sure that the actions and behavior of the individual functions are correct. The description in this chapter is therefore restricted to these aspects.

usropn is invoked first of all once, and once only, for each assigned file. The workio argument causes a work area of 1024 bytes to be made available as file-specific memory. This area is adopted automatically for all subsequent calls until the usrcls.

The openmode argument specifies the desired access type (input, output). The record_format, record_size, etc. arguments specify the file and record attributes, which can be adapted to each particular file if necessary.

The successful termination of the function and any special states or errors can be reported by means of predefined and freely assignable return codes. Each return code is evaluated by FLAM, and passed on to the invoking programs if an error is established.

A record is transferred for writing with usrput. If the record cannot be written with the specified size, the return code must include the information that it has been either shortened or padded with the characters specified by padchar in the usropn.

FLAM requests the next record with usrget. The maximum number of characters which can be transferred is specified in the buffer_length parameter. If the record needs to be shortened as a result, this must be indicated by the return code.

The return code must also indicate when the end of the file is reached. The record size must be returned for each record which is read (even if the record format is fixed).

usrcls causes the file to be closed. The work area for this file is released again by FLAM after control has been handed back.

urspos is a dummy function in UNIX.

1.5.5 The User Exits

A user exit is a user-provided program that is invoked by FLAM at specific events during the processing via the defined interface.

FLAM supports two catagories of user exits: file access exits and an exit for automatic key management.

1.5.5.1 File Access Exits

These exits permit the user to postprocess the records output by FLAM (both those contained in the FLAMFILE and those in the decompressed file) before they are finally saved, i.e. they can be modified, deleted or given additional records. The records which are read by FLAM can be prepared for processing by FLAM in the same way.

The term "access" refers in this context either to an input/output instruction or to a corresponding call of the user-defined input/output, and not to a call of a record interface function. Thus - at least as far as the FLAMFILE is concerned - the user exits are not synchronized with the application program, since the latter is not able to establish, for example, whether or not a flmget call actually does initiate one or more read operations.

These user exits can be activated via the subprogram interface or the record interface, the latter being able to activate the user exits for FLAMFILE accesses only. The programs invoked through these interfaces must be staticly linked to the application (see section 6.2).

A user exit is invoked each time the file is opened, read, written or closed. The functioncode argument indicates to the user exit the access type, to permit it to respond in the appropriate manner to each particular situation.

FLAM supports the following user exits for file accesses:

User exit	Invoked for accesses to
exk10	Original file (compress)
exk20	FLAMFILE (compress)
exd10	Decompressed file (decompress)
exd20	FLAMFILE (decompress)

User exits exk10 and exk20 can be used during the compression procedure for accesses to the original file or to the FLAMFILE, whereas exd10 and exd20 can be used during the decompression procedure to access the decompressed file or the FLAMFILE. If records contained in the FLAMFILE were modified by user exit exk20 when

they were compressed, they can only be decompressed if the modifications are undone again by the complementary user exit exd20, i.e. if every FLAM record is reset by exd20 to the state in which it was originally transferred by FLAM at user exit exk20.

1.5.5.2 User Exit for Automatic Key Management

Automatic key management facilitates handling encrypted files by relieving the user from duties like generating, exchanging and storing passwords, in whatever way.

When invoking the key management exit routine for encryption, FLAM also passes an optionally user-supplied character string via the interface. The exit returns a password which is used by FLAM for encrypting. In addition, the exit may also return a byte string that FLAM attaches to the encrypted FLAMFILE. When decrypting, FLAM passes this string back to the exit for the retrieval of the password.

This exit can be activated by the flam command or the subprogram interface. It is not available via the record interface. Programs invoked through these interfaces must be created as shareable objects and are linked dynamically (see section 6.2).

Note: Throughout this manual the terms "password" and "(encryption) key" are used synonymously.

FLAM (UNIX)

User Manual

Chapter 2:

The flam Command

2. The flam Command

2.1 Functions

The following functions can be invoked with the flam command:

- File compression with or without encryption (using the -compress parameter)
- File decompression with decryption, if needed (using the -decompress parameter)
- Modification of installation-specific default values (using the -defaults parameter)
- List of installation-specific default values (using the -list parameter)

When the flam command is invoked, all the necessary information can be specified at different levels, namely explicitly in the command itself, in a parameter file or implicitly by means of default values. The description of the command syntax below sets out the logical relationships between the syntax elements, in other words between the parameters and their arguments, irrespective of the level at which they are actually specified.

The classification of the syntax elements in this chapter as either mandatory or optional applies to this FLAM check. The syntax rules consequently refer to all the settings together, and not simply to the inputs in the command line.

Other than in relation to the priority rules, the specification level is therefore not relevant to the syntax description. Thus, where the explanations below refer to parameters, they apply equally irrespective of whether they are taken to mean the flam command, the parameter file or the default values. Explicit reference is made to exceptions.

When the flam command is entered, either upper or lowercase notation can be used for FLAM, FLAMFILE and all the other parameter names.

2.2 Command Syntax

The shell command flam invokes the utility either interactively or from a shellscript.

Syntax

FLAM -parameter[=*value*] [...]

Since the parameters can be specified in a parameter file as well as in the command, and since the missing parameters necessary to execute a particular function must then be taken from the default value file, the order of the specifications is arbitrary.

Parameters and their values may be abbreviated by omitting as much of their endings as possible without causing ambiguity. For example, -recs may be used for -recsize, while using less characters could be taken to mean -recdelim, -recformat, or -recsize.

The parameters which are allowed for each function are specified in the appropriate sections of this chapter. A detailed description of all the parameters in alphabetical order can be found in section 2.3.

For better redability, all parameters in this manual are preceded by a minus sign ("-"). Its use, however, is optional.

Notational remark:
On this and the following pages, square brackets ("[" and "]") denote syntax elements optional in the respective contexts. Command parts printed in *italics* are to be substituted by the user for actual values. Ellipses stand for repetitions of the previous syntax element.

2.2.1 FLAM Compression

Syntax flam -compress -parameter[=*value*] [...]

-attributes = *attribute option*

-compress

-flamcode = c*haracter set*

-flamfile = *file specification*

-flamin = *file specification*

-inrecdelim = *record delimiter*

-inrecformat = *record format*

-inrecsize = *n*

-kmexit= *exit specification*

-maxbuffer = *n*

-maxrecords = *n*

-msgfile = *message file*

-mode = *compression mode*

-ndc

-nopath

-parfile = *parameter file*

-password = *password*

-recformat = *record format*

-recsize = *n*

-show = *display option*

-translate = *code table*

2.2.2 FLAM Decompression

Syntax flam -decompress -parameter[=*value*] [...]

-flamfile = *file specification*

-flamout = *file specification*

-kmexit= *exit specification*

-msgfile = *message file*

-option = *FLAM option*

-outpath = *output path*

-outrecdelim = *record delimiter*

-outrecformat = *record format*

-outrecsize = *n*

-pad_char = *padding character*

-parfile = *parameter file*

-password = *password*

-show = *display option*

-translate = *code table*

2.2.3 Displaying the FLAM Default Values

Syntax

FLAM -list[=*list specification*]

See description of the -list parameter.

2.2.4. Setting the FLAM Default Values

Syntax

FLAM -defaults=*default specification*

This call can only be used by the system manager. See description of the -defaults parameter.

2.3 flam Command Parameters

This section describes the parameters of FLAM in alphabetical order, irrespective of the functions with which they can be used.

-attributes

This parameter causes original file information to be entered in the FLAMFILE.

Syntax

-attributes=*attribute option*

Values

attribute option	**Meaning**
none	No file information
common	Common file information
all	Common file information and system-specific information about the original file

Description

Entering file information in the FLAMFILE permits it to be extracted later on without having to decompress the entire FLAMFILE. The organization, the record format and the record size of the original file can be saved during the compression procedure, together with a flag which identifies the operating system in which the compression took place (-attributes=common).

Thus, when the file is decompressed, it can be restored with its original characteristicsif necessary.

The original file specification can be entered optionally (-attributes=all). When the file is decompressed, the output specification [ ] or [*] can then be set, so that FLAM automatically restores both the entered file specification and the associated characteristics.

If -attributes=none, no information about the original file is entered in the FLAMFILE.

-compress

This parameter causes FLAM to compress the original file.

Syntax

-compress

Values

None

Description

When FLAM is invoked, -compress must be specified in order to compress the original file and create a FLAMFILE.

-decompress

This parameter causes FLAM to decompress the compressed file.

Syntax

-decompress

Values

None

Description

When FLAM is invoked, -decompress must be specified in order to decompress the compressed file and create a decompressed file or show the information about the original file(s).

-defaults

This parameter permits the installation-specific default values to be modified. It can only be used by the system manager.

Syntax

-defaults=*default specification*

Values

default specification

Default specifications always consist either of a keyword or of a keyword with an assigned value. With only a few exceptions, the keywords are identical to the parameters of the flam command and the assigned values are subject to the same syntax as for these parameters.

Where the default specifications described below include references to the corresponding parameters, their syntax and meaning can be taken from the descriptions of these parameters.

ascii_ebcdic=*translation option*

This specifies how individual ASCII characters are to be converted to EBCDIC code. Translation options have the following format:

ascii_code:ebcdic_code

ascii_code and ebcdic_code are the codes for the ASCII character which must be converted and for the EBCDIC character which must be inserted; they are both specified in the form of a hexadecimal number between 00 and ff.

Example:

-defaults=ascii_ebcdic=41:81

causes each ASCII "A" to be converted to an EBCDIC "a" during a compression procedure for which -translate=e/a has been specified.

attributes=*attribute option*

See description of the -attributes parameter.

Note:

It is advisable to set -attributes=all as the default value, to ensure that FLAMFILES which have been created using the default values, and which contain compressed versions of several original files, can be decompressed into individual files again with the original names and the original attributes, such as the record format. Otherwise, this information will be lost when the default values are used and it will not be possible to reconstruct the origin of the files again later on.

compress

See description of the -compress parameter.

decompress See description of the -decompress parameter.

ebcdic_ascii=
translation option This specifies how individual EBCDIC characters must be converted to ASCII codes.

Translation options have the following format:

ebcdic_code:ascii_code

ebcdic_code and ascii_code are the codes for the EBCDIC character which must be converted and for the ASCII character which must be inserted; they are both specified in the form of a hexadecimal number between 00 and ff.

flamcode=*character set* See description of the -flamcode parameter.

flamfile=*file specification*
flamfile=none This defines the default name for the FLAMFILE. The file specification must name a single file uniquely; it must not contain any patterns, nor must it be specified as a substitution rule.

Note that when a default name is defined, stdin, stdout, and pipelines can no longer be used for flamfile.

With flamfile=none, the default value for this parameter is dropped.

This file name is the default value for the output file during the compression procedure and the default value for the input file during the decompression procedure. The characteristics which are specified with recformat, recsize, recdelim, delete, user_io, exd20 and exk20 all refer to this file.

flamin=*input specification*
flamin=none This defines the default name for the input file during the compression procedure. The input specification is a file specification. It must name a single file uniquely an may not contain patterns (wildcards).

Note that when a default name is defined, stdin and pipelines can no longer be used for flamin.

With flamin=none, the default value for this parameter is dropped.

flamout=
output specification
flamout=none

This defines the default name for the output file during the decompression procedure. The output specification is a file specification. It must name a single file uniquely and must not contain any patterns(wildcards) nor a substitution rule.

Note that when a default name is defined, stdout and pipelines can no longer be used for flamout.

With flamout=none, the default value for this parameter is dropped.

inrecdelim See description of the -inrecdelim parameter.

inrecformat See description of the -inrecformat parameter.

inrecsize See description of the -inrecsize parameter.

maxbuffer See description of the -maxbuffer parameter.

maxrecords See description of the -maxrecords parameter.

mode=*FLAM mode* See description of the -mode parameter.

msgfile=*message file* See description of the -msgfile parameter.
msgfile=none With msgfile=none, any default value for this setting is dropped.

option=*FLAM option* See description of the -option parameter.

outrecdelim See description of the -outrecdelim parameter.

outrecformat See description of the -outrecformat parameter.

outrecsize See description of the -outrecsize parameter.

parfile=*parameter file* See description of the -parfile parameter.
parfile=none With parfile=none, any default value for this setting is dropped.

recdelim See description of the -recdelim parameter.

recformat See description of the -recformat parameter.

recsize See description of the -recsize parameter.

show=*display option* See description of the -show parameter.

translate=*code table* See description of the -translate parameter.
translate=none With translate=none, any default value for this setting is dropped.

Description

The system manager can define default values for all the parameters of the flam command except the -defaults, -list, -kmexit, -ndc, -nopath, -outpath, and -password parameters. The default values can only be modified by a command.

The default values can be defined by invoking FLAM with the -defaults parameter and specifying the default specification. This consists of a keyword and possibly an assigned value. The syntax of the default specifications is the same as that of the flam parameters. In addition, there are also the ascii_ebcdic and ebcdic_ascii parameters.

When defining default settings one should take into account that some settings cannot be modified by the user via command line or parameter file entries. This is the case, for example, when –defaults=flamin=... is defined. The name of the file may still be modified but there is no way to use stdin as input.

-flamcode

This parameter defines the character set for character-coded information in the FLAMFILE when a file is compressed, such as the file name and control characters.

Syntax

-flamcode=*character set*

Values

character set	**Meaning**
ascii	Use ASCII code for the FLAMFILE.
ebcdic	Use EBCDIC code for the FLAMFILE.

Description

This parameter specifies the character set which is to be used to represent character-coded information in the FLAMFILE during the compression procedure. In adc, cx8, and vr8 modes, this only concerns the information contained in the FLAM file header, such as the original file name and a few control characters, since the compressed file is represented in binary form. In the cx7 mode, all the FLAM control characters in the compressed file are also coded in this character set. The characters extracted from the original data, however, remain unaffected in all modes. Their translation can be achieved by the -translate parameter.

-flamfile

This parameter specifies the FLAMFILE(s).

Syntax

-flamfile=*file specification*

Values

file specification

With compression, *file specification* specifies a specific file or a list of files separated by commas. A substitution rule is also permitted (see Sect. 2.4).

With decompression, *file specification* specifies a specific file, a pattern (with wildcards * or ?), or a list of such elements separated by commas.

Description

During compression, the compressed data is written in the specified file(s).

During decompression, the compressed data is read from the specified file(s).

If the *file specification* contains more than one file, the rules described in sections 2.4.1 to 2.4.3 with regard to the syntax of the file specification and the assignment to the *input specification* (compression) or the *output specification* (decompression) must be observed.

-flamin

This parameter specifies the files to be compressed.

Syntax

-flamin=*input specification*

Values

input specification

The *input specification* specifies a specific file, a directory, a search pattern (with wildcards * or ?), or a list of such elements separated by commas.

Description

The specified files are compressed in accordance with the other parameters specified for the command. When a name refers to a directory, all files in that directory and, if present, its subdirectories are compressed.

If the *input specification* contains more than one element, the rules described in sections 2.4.1 and 2.4.3 regarding the syntax and the assignment to the FLAMFILE specifications must be observed.

-flamout

This parameter specifies the names of the decompressed files.

Syntax

-flamout=*output specification*

Values

output specification

The *output specification* specifies a specific files, a search pattern (with wildcards * or ?), or a list of such elements separated by commas. It may also consist of one of the special output specifications [ ], [*], or [*fileame*] (see Sect. 2.4).

Description

During the decompression procedure, the decompressed data of the FLAMFILE is written in the specified file(s).

If the *output specification* contains more than one file, the rules described in sections 2.4.2 and 2.4.3 with regard to the syntax and the assignment to the FLAMFILE specifications must be observed.

-inrecdelim

This parameter specifies a record delimiter for original files.

Syntax

-inrecdelim=*record delimiter*

Values

record delimiter

Hexadecimal characters between 01 and ff, with a length of either 1 or 2 bytes.

Description

This parameter specifies the record delimiter for an original file with a stream record format. If a stream record format is specified without a record delimiter, 0x0a and 0x0d0a are interpreted as the end of a record.

If only 0x0a should be interpreted as the end of a record, -inrecdelim=0a must be specified, in order to ensure that occurrences of 0x0d which precede 0x0a are interpreted as part of the data and not as part of a record delimiter. If only 0x0d0a should be interpreted as a record delimiter in a particular file, -inrecdelim=0d0a must be specified, to ensure that any 0x0a strings which occur alone are identified as belonging to the data.

Any other character combinations can be used as record delimiters, in addition to 0x0a and 0x0d0a.

-inrecformat

This parameter specifies the record format of the original file.

Syntax

-inrecformat=*format option*

Values

format option	Meaning
eaf	EAF data, variable record size with a 4-byte size field in ASCII or EBCDIC code. The record size only specifies the length of the data excluding the size field. The size field is part of the data.
fix	Fixed record size as set by -inrecsize.
stream	Variable record size (max. 2048 characters) with record delimiter. Typically text data.
undefined	Records with no defined structure; records of an identical size are read (with the possible exception of the last one).
variable	Variable record size with 2-byte binary record size (including length field).
var_2b	Variable record size with 2-byte binary record size (including length field).
var2b_data	Variable record size with 2-byte binary record size (**not** including length field).
var_4b	Variable record size with 4-byte binary record size (including length field) (host).
var_ascii	Variable record size with 4-byte size field in ASCII code (including length field).
var_ebcdic	Variable record size with 4-byte size field in EBCDIC code (including length field).

Description

This parameter specifies the record format of the original file. The data is read from the file in the form of logical records with the specified format.

-inrecsize

This parameter specifies the record size of the original file.

Syntax

-inrecsize=*n*

Values

n

Integer number between 1 and 32760

Description

This parameter specifies the record size of original files with a fixed or undefined record format.

-kmexit

This parameter specifies a user exit routine for automatic key management.

Syntax

-kmexit=*exit specification*

Values

exit specification has the general form

[(]*func*[([*lib*])[*exparm*]][)]

where

func is the name of a function in a shared library that supplies the password (key) needed for en/decryption.

lib is the name of a shared library (without the .so suffix) containing *func*. When *lib* is omitted, FLAM loads the function from **libflamkm.so** which it expects to find in directory $FLAM_PATH/../lib when that environment variable is defined, or otherwise in /usr/lib.
Note: When entered in a shell command, each parenthesis must be preceded by an escape character (\) to avoid its being interpreted by the shell. Escape characters are not requitred in parameter files.

exparm is a sequence of up to 256 characters to be passed to the password function *func*. It must not contain space characters or parentheses. If it contains commas, the entire *Exit specification* must be enclosed in parentheses. *exparm* must always be preceded by a pair of parentheses which may be empty when *lib* is omitted.

Description

With this parameter, FLAM is told the name of a function it has to invoke in order to receive a password needed for encryption or decryption of a file. This function must reside in a shared library (file type .so) the name of which may be specified after the function name. After the library name, an alphanumeric string may be appended that will be passed as a parameter string to the invoked function.

No default value can be set for -kmexit.

This parameter and -password are mutually exclusive.

For details on the interface to this user exit and its building see section 6.2.

-list

This parameter permits a list of the installation-specific default values to be output. The list can only be displayed using a command.

Syntax

-list[=*list specification*]

Values

list specification

Any flam parameter for which a default value can be set.

Description

The installation-specific default values can be displayed with the list parameter.

When a *list specification* is supplied, the current setting of the default value for the respective paramter is displayed.

When *list specification* is omitted, a list of all current default values is displayed except the code tables for ASCII-EBCDIC and EBCDIC-ASCII translations. Also, the FLAM version and its creation date are displayed.

The details are displayed similar to the following example while the operating system, the creation date and the effective values correspond to those of the particular installation.

```
FLAM (R) 4.2.0 for Linux
copyright (c) 2012 by limes datentechnik gmbh
Build from Aug  2 2012 - 14:53:25
Expiration date: 12/31/2012
Licensed for limes_datentechnik_gmbh
-------------------------------------------
             Default  settings
-------------------------------------------

attributes ....................... all

decompress

flamcode ......................... ASCII

flamfile (compressed file) ........ (not specified)
              recformat ........... fix
              recsize ............. 512

flamin (source file) .............. (not specified)
              inrecformat ......... stream
              inrecdelim .......... x'0a'

flamout (decompressed file) ....... (not specified)
              outrecformat ........ stream
              outrecdelim ......... x'0a'

maxbuffer ........................ 32768

maxrecords ....................... 4095

message file ..................... (not specified)

mode ............................. adc
   encryption ..................... none

option ........................... nocut
                                   nosuppress

parameter file ................... (not specified)

show ............................. all

translate (conversion file) ....... (not specified)
```

The *list specifications* ascii_ebcdic and ebcdic_ascii cause the respective code tables to be displayed. The format of the display is the same as described for the setting of these default values (see description of the parameter -defaults).

-maxbuffer

This parameter specifies the size of the matrix buffer.

Syntax

-maxbuffer=*n*

Values

n

Integer number between 1 and 2,621,440. Values *n* where $0 \leq n \leq 2{,}560$ are interpreted as a number of Kbytes (1 Kbyte = 1,024 bytes), while higher values are interpreted as a number of bytes. FLAM chooses one of the buffer sizes from the table below (in Kbytes):

2	4	6	8	10	12	14	16
32	48	64	80	96	112	128	144
176	224	256	288	320	352	384	416
512	640	768	896	1.024	1.536	2.048	2.560

If the specified value is not contained in the table, the nearest higher buffer size is taken instead (if any); otherwise, the maximum value is always used (2,560).

Description

During the compression with modes cx7, cx8, and vr8, FLAM reserves the matrix buffer for the records of the original file. Please note that a matrix buffer of the same size must also be made available for the decompression procedure, even if this takes place on a different system.

With mode adc, this parameter has no effect.

-maxrecords

This parameter specifies the maximum number of records which can be compressed in the same block.

Syntax

-maxrecords=*n*

Values

n

Integer number between 1 and 4095.

Description

FLAM buffers records in the matrix buffer up to the specified number and compresses them. This specified number may occasionally not be reached if the matrix buffer is too small for the full number of records. The compression efficiency generally increases with the number of records. If -maxrecords=1, the compression is sequential and record-oriented.

With modes cx7, cx8, and vr8, up to 255 records are stored in one matrix. If a higher value is specified it takes effect only with mode adc.

-mode

This parameter specifies the mode in which the original file is compressed and, when desired, encrypted.

Syntax

-mode=*FLAM mode*

Values

FLAM mode

FLAM mode	Meaning
adc	With -mode=adc the input file is compressed in the adc mode and a binary FLAMFILE is created. This is the most efficient mode and can be used universally irrespective of the data structure. With this mode, a password can be specified to encrypt the FLAMFILE.
aes	-mode=aes causes the input file to be compressed in the adc mode with AES encryption. A password must be specified with the -password parameter.
flamenc	-mode=flamenc causes the input file to be compressed in the adc mode with FLAM encryption (the proprietary encryption method used with FLAM 3). A password must be specified with the -password parameter.
cx7	With -mode=cx7, the input file is compressed in the cx7 mode and a character-coded FLAMFILE is created. However, files should only be compressed in this mode if they do not contain any non-printing characters. It is a slightly less efficient mode, though on the other hand the FLAMFILE which is created can be converted to other character sets, e.g. from ASCII to EBCDIC, without losing information.
cx8, vr8	With -mode=cx8 or -mode=vr8, the input file is compressed in the cx8 or vr8 mode, respectively, and a binary FLAMFILE is created. The purpose of these two modes is to ensure compatibility with previous versions.

Description

The input file is compressed in the specified FLAM mode and, when a password is entered, encrypted with the selected method. When mode adc is specified and a password is given, the result is the same as with -mode=flamenc.

-msgfile

This parameter can be used to redirect FLAM message output to a file.

Syntax

-msgfile=*message file*

Values

message file

This specifies a file to which the FLAM messages must be written.

Description

FLAM messages are output to stderr which is generally the user's screen when working interactively. With -msgfile=*message file*, output can be sent to a file to act as a permanent copy for documenting the compression procedure. This is important for batch processing, for example.

However, logging in the message file does not begin until after the syntax and compatibility of the FLAM settings specified in the command have been checked. Error messages concerning syntax errors, missing files or invalid settings are not logged. In such cases, FLAM stops processing without creating a message file.

-ndc

This parameter can be used in adc mode to suppress compression.

Syntax

-ndc

Values

None

Description

For data that is known to compress badly, FLAM's performance can be improved by this parameter which suppresses compression. With modes aes and flamenc, only encryption is done while with mode adc this parameter causes FLAM to simply copy the data.

With other modes, -ndc has no effect.

FLAMFILEs created with -ndc are always Secure FLAMFILEs and are logged as adc-compressed files at decompression.

-ndc cannot be set as default.

-nopath

This parameter suppresses including information on the directory path of the original file in the compressed data.

Syntax

-nopath

Values

None

Description

When compressing with –attributes=all, FLAM stores the entire absolute path name of the original file in the compressed file's file header. Trying to create a file with the original path at decompression may cause problems on systems with a different directory structure or file naming syntax. With the –nopath parameter, only the filename proper is stored while the directory path is discarded.

When decompressing files compressed with –nopath, using –flamout=[*] has the same effect as –flamout=[ ].

-nopath cannot be set as default.

-option

This parameter controls FLAM's behaviour when decompressed records are output.

Syntax

-option=*FLAM option*

Values

FLAM option

FLAM option	**Meaning**
cut	Records with a length greater than the maximum record size are truncated.
nocut	Records with a length greater than the maximum record size are not truncated.
suppress	Blank space at the end of a record is suppressed.
nosuppress	Blank space at the end of a record is not suppressed.

Description

During the decompression, the record format and size of the decompressed file can either be specified explicitly or adopted from predecessor versions, so that the maximum record size is less than in the original file. If the cut option is specified, the decompression continues and any records which are too long are truncated to the maximum size. If nocut is specified, decompression is aborted and an error message is output.

If the decompressed file is written with the stream record format during the decompression procedure, blanks at the end of a record can be suppressed by specifying suppress. nosuppress causes the blanks to be written.

-outpath

This parameter determines the directory location used by FLAM to create a decompressed file.

Syntax

-outpath=*output path*

Values

output path

This specifies an existing directory at which FLAM will create the file(s) being decompressed. This may be an absolute or a relative path.

Description

An explicit specification –flamout=... at decompression may also include a complete directory path. This option is not available when decompressing with –flamout=[ ]. In that case, an output directory can be specified using -outpath=... .

No default value can be set for -outpath.

-outrecdelim

This parameter specifies a record delimiter for decompressed files.

Syntax

-outrecdelim=*record delimiter*

Values

record delimiter

Hexadecimal characters between 01 and ff, with a length of either 1 or 2 bytes

Description

This parameter specifies the record delimiter for a decompressed file with the stream record format.

If a record delimiter is not specified for the stream record format, 0x0a is used as a default delimiter. If outrecdelim =0d0a is specified, the decompressed file can be generated in MS-DOS format.

Any other character combinations are also allowed as record delimiters.

-outrecformat

This parameter specifies the record format of the decompressed file.

Syntax

outrecformat=*format option*

Values

format option

See description of *format options* for -inrecformat.

Description

This parameter specifies the record format of the decompressed file. The decompressed data is written in the file in the form of logical records with the specified format.

-outrecsize

This parameter specifies the record size of the decompressed file.

Syntax

-outrecsize=*n*

Values

n

Integer number between 1 and 32760

Description

This parameter specifies the record size of the decompressed files with a fixed or undefined record format. For other record formats, this indicates rhe buffer size.

-pad_char

This parameter specifies the character which is used to pad the records in the decompressed file.

Syntax

-pad_char=*hexcode*

Values

hexcode

Hexadecimal characters between 00 and ff

Description

During the decompression procedure, the record format and size of the decompressed file can be specified explicitly, so that the record size is greater than in the original file. In this case, the records are padded to the specified length.

-parfile

This parameter permits the specifications in the command line to be supplemented or overridden by FLAM parameters in a parameter file.

Syntax

-parfile=*parameter file*

Values

parameter file

This parameter specifies a file containing FLAM parameters. When no path is given, the current directory is searched.

Description

Specifying a parameter file permits default values to be overridden without the need for complex inputs in the command line (see also section 2.5, Priority Rules for FLAM Settings). This file can be created interactively using a text editor (see example in Sect. 2.6).

One parameter file cannot refer to another, i.e. it may not contain a parfile specification. Furthermore, it may not change the –show parameter whose setting is determined when the command line is analyzed.

-password

This parameter is used to specify a password for encrypting or decrypting a FLAMFILE in aes or flamenc mode when compressed with mode adc.

Syntax

-password=*password*

Values

The value for *password* can be entered in two formats

- As a simple character string

  In this format the password is entered immediately following the equal sign as a string of up to 64 characters. Valid characters are all capital and small Latin letters, decimal digits, the minus sign (-) and underscore (_). Using any other characters may yield unpredictable results. Note that capital and small letters are different characters.
  Example: -password=Alligator

- As a sequence of hexadecimal digits

  In this format the password may consist of an even number of up to 128 hexadecimal digits (0-9, a-f, A-F). In this context small and capital letters are interchangeable. The sequence is enclosed in single quotes (\‘) and preceded by the letter “X“. Any such combination is permitted.
  Example: -password=X\‘416c6c696761746f72\‘

Description

FLAM uses the specified password during compression with mode adc to encrypt the compressed data and during decompression to decrypt the encrypted compressed data. The passwords used at compression and decompression must be identical or else the decompression command is rejected with an appropriate error message.

With modes other than adc, aes, or flamenc this parameter causes an error. When used with mode adc, the flamenc method is used for encryption.

A password entered as a character string is equivalent to a hexadecimal sequence consisting of its character codes and vice versa.

Note that a given character string entered on an ASCII system is not equivalent to the same string entered on a system using EBCDIC code. Therefore, for data exchange among systems with different character codes the hexadecimal format must be used if the codes of the password characters are not available on the keyboard.

This parameter cannot be set by default.

This parameter and -kmexit are mutually exclusive.

-recdelim

This parameter specifies the record delimiter for the FLAMFILE.

Syntax

-recdelim=*record delimiter*

Values

Hexadecimal characters between 01 and ff, with a length of either 1 or 2 bytes

Description

This parameter permits a record delimiter to be specified for the stream record format.

If a record delimiter is not specified for the stream record format, 0x0a is used as a default delimiter.

If -recdelim=0d0a is specified, for example, the data can be output in MS-DOS format.

This parameter can only be specified for the compression procedure.

-recformat

This parameter specifies the record format of the FLAMFILE.

Syntax

-recformat=*format option*

Values

format option

format option	**Meaning**
fix	Fixed-size records, size as set by the -recsize parameter.
var	Variable-size records; the record size field is 2 bytes long
stream	Variable-size records with a record delimiter

Description

The FLAMFILE always contains records of identical size.

In the cx8 and vr8 modes, the records can be written either with (variable) or without (fixed) a record size field.

The stream record format is used in the cx7 mode. If -recdelim is not specified, 0x0a is used as the default record delimiter.

recformat need only be specified for the compression procedure.

-recsize

This parameter specifies the record size of the FLAMFILE.

Syntax

-recsize=*n*

Values

n

Integer number between 80 and 32760 for all modes except cx7

Integer number between 80 and 4095 for the cx7 mode

Description

This parameter specifies the record size of the FLAMFILE.

The compressed data is always written in records of the same size, irrespective of the record format.

There is no relationship between the compressed blocks and the records in the compressed file. A record may contain data pertaining to one or more compressed blocks. A compressed block may be contained in one or more records.

There is no relationship between the records in the compressed and non-compressed files.

This parameter need only be specified for the compression procedure.

-show

This parameter specifies which information should be shown by FLAM.

Syntax

-show=*display option*

Values

Display option

display option	**Meaning**
none	No information is shown
all	All the available information is shown. This depends on the invoked operation (compress or decompress), but not on whether FLAM was invoked directly or using the parameter file. The following information is shown during the compression procedure: FLAM version FLAM settings Operation (compress) Code table Compression mode Matrix buffer size Maximum number of records/matrix buffer Character set (ASCII or EBCDIC) attributes setting Name of the original file Formatting information for the original file Name of the compressed file (FLAMFILE) Formatting information for the compressed file Statistical information (see show=statistic) Error messages and warnings

display option	**Meaning**
	The following information is shown during the decompression procedure: FLAM version FLAM settings: Operation (decompress) Code table (only if the FLAM call includes the necessary specifications) Name of the compressed file (FLAMFILE) Name of the decompressed file Formatting information for the decompressed file Saved compression information (see show=attributes) Statistical information (see show=statistic) Error messages and warnings
attributes	(Decompression only) This suppresses creation of the decompressed file Only the saved compression information is shown Name of the original file (only if -attributes=all was specified when it was compressed) Formatting information for the original file Compression mode Character set (ASCII or EBCDIC) System which created the FLAMFILE
error	Only error messages and warnings are shown.

display option	**Meaning**
statistics	Error messages, warnings and the following statistics are shown: Number of compressed records Number of compressed bytes Number of non-compressed records Number of non-compressed bytes Compression efficiency as compared with the original file (compression only) [1] Runtime [1] This efficiency is the ratio of the number of bytes actually read to the number of bytes output and is specified as a percentage. Record size fields, record delimiters and characters for padding compressed records are not taken into account.

Description

This parameter controls the information output by FLAM. You can choose between detailed information (all), error messages and warnings either with or without statistics (error or statistic) and no information (none).

During the decompression procedure, you can decide just to show the information about the original file, without creating a decompressed file. Use -show=attributes to do so. If the FLAMFILE contains compressed versions of several files, the information about each of the original files is shown one file at a time. However, this information is only shown if this was specified when the files were compressed into the FLAMFILE (see -attributes parameter). If -show=attributes is specified, the output file parameter is ignored.

Message outputs can be redirected to a file by specifying 2>>*filename*. If -msgfile=*filename* is specified, on the other hand, the FLAM messages are written into a file, but system messages continue to be displayed on the screen.

-translate

This parameter causes the codes for the data in the original file to be translated prior to the compression procedure or those for the data in the decompressed file to be translated after the decompression procedure.

Syntax

-translate=*code table*

Values

Code table

This specifies a file containing the code table. The code table is a string of 256 characters making up the character set into which the data is to be translated. (Please refer to Appendix C, Code tables, for further details of the code table.)

Description

If -translate=*code table* is specified for the compression procedure, every record in the original file is translated according to the specified code table before it is buffered in the matrix buffer. During the decompression procedure, this specification causes every decompressed record to be translated according to the code table before it is inserted in the decompressed file.

When FLAM is installed, code tables for converting from ASCII to EBCDIC and from EBCDIC to ASCII are made available as standard.

If one of the code tables in the default value file is to be used, the following specifications are necessary:

a/**e** to convert from ASCII to EBCDIC,
e/**a** to convert from EBCDIC to ASCII

2.4 File Specifications

File specifications are subdivided into input specifications and output specifications.

The input specification for the compression procedure (original file) is specified by -flamin, and the output specification (compressed file) by -flamfile.

The input specification for the decompression procedure (compressed file) is specified by -flamfile, and the output specification (decompressed file) by -flamout.

2.4.1 Input Specifications

The input specification may consist of a file or directory name, a pattern (with wildcards), or a list of such elements. Elements of a list are separated by commas.

Filenames may contain space characters, but not as first characters. Filnames containing spaces must be enclosed in double quotes.

The files specified as input files must actually exist. At least one matching file must exist for a pattern. Otherwise, the flam command will be aborted and an error message output.

If the input specification specifies files for processing by FLAM, either the -flamin parameter or -flamfile must be specified for the compression or decompression procedure respectively.

If the data is to be entered via the special stdin file, the parameter for the input specification is omitted. FLAM can thus be used as a filter.

2.4.2 Output Specifications

The output specification may consist of a filename, a list of filenames (separated by commas), or a substitution rule. A substitution rule has the following format:

[*string1*=*string2*]

Moreover, one of the output specifications [ ], [*], or [*filename*] may be used (see description below).

Output files are created if they do not exist and overwritten if they do.

If a substitution rule is defined as an output specification, the name of the FLAMFILE or the decompressed file must be formed by substituting *string2* for *string1* in the name of the input file. The strings and the equal sign must have no separating spaces in between. The substitution rule applies only to file names and not to path names.

Example:

```
flam –compress –mode=cx8
–flamin=helptxt1.txt,helptxt2.txt
–flamfile=[.txt=.cmp]
```

causes the files called helptxt1.txt and helptxt2.txt to be compressed and the compressed data to be saved in the associated FLAMFILEs called helptxt1.cmp and helptxt2.cmp.

The substitution rule must be applicable to all the assigned input files, i.e. *string1* must be contained in the names of every one of these files, or the flam command will be aborted for the file to which it does not apply and an error message will be output. Only one substitution takes place when the name of the output file is formed, even if *string1* occurs more than once in the name of the input file.

The output specifications [*file name*], [ ], and [*] have special meanings.

Using [*file name*] as output specification causes the decompression to select one or - when *file name* contains wildcards -, possibly several, matching file(s) from a FLAMFILE archive. Those are decompressed into the current directory unless –outpath=... directs otherwise.

If the output specification for the decompression procedure is [*], the names of the decompressed files are derived from the FLAMFILE, provided the FLAMFILE actually contains them. If not, an error message is output. -attributes=all must be specified for the compression procedure, in order to include the names of the original files in the FLAMFILE.

If the output specification is [], the effect is similar to the output specification [*], i.e. the names of the decompressed files are derived from the FLAMFILE(s), except that only the file name is taken into account. The path name is ignored. All the files are thus created in the current directory if no –outpath specification is given.

Once again, -attributes=all must have been set for the compression procedure.

The output specifications [*file name*], [], and [*] must not be part of a list of output specifications, i.e. they must stand alone.

If the output specification specifies files for processing by FLAM, either the -flamfile parameter or -flamout must be specified for the compression or decompression procedure respectively.

If the data is to be output to the special file stdout, the parameter for the output specification must be omitted. FLAM can thus be used as a filter.

2.4.3 Assignment of Input Specifications to Output Specifications

If the flam command contains several different input and output specifications, they are assigned to one another. The assignment determines where the compressed data of each original file and the decompressed data of each compressed file is saved.

The input and output specifications are assigned to one another by means of their positions in the lists. The first input specification is assigned to the first output specification and the second input specification to the second output specification, etc. If there are more input specifications than output specifications, the excess input specifications are assigned to the final output specification. If there are too many output specifications, the excess ones are ignored.

If the output specification is a file specification, all the outputs which result from processing the assigned input files are saved in this file. The file attributes must be compatible.

If the output specification is a substitution rule, a separate FLAMFILE, which is named according to this rule, is created during the compression procedure for each of the assigned input files.

During the decompression procedure, the substitution rule is applied to the names of the FLAMFILEs and not to the names of the compressed original files, i.e. each FLAMFILE results in a separate decompressed file.

If a FLAMFILE which contains the compressed data of several different files is assigned an output file, all the compressed data in this FLAMFILE is decompressed into the output file. The output specification [] or [*] is then necessary in order to create separate files again. In this case, each compressed block is decompressed into a file with the original name.

In the example below, the flam command contains a list of input specifications and a list of output specifications.

```
flam -compress -attributes=all
-flamin=*.dat,*.txt,func1.hlp,func2.hlp
-flamfile=[.dat=.cmp],text.arc,[hlp=xyz]
```

The list of input specifications contains four entries, namely the output specifications, two substitution rules and a file specification. In accordance with the assignment rules, a file with the same name and a .cmp extension is created for each file which has a .dat extension, and the compressed data of all the files with a .txt extension is saved in a single FLAMFILE called text.arc. The compressed data of func1.hlp is saved in func1xyz on account of the substitution rule contained in the final output specification. Since there are no further output specifications, this rule is also applied to func2.hlp, i.e. a FLAMFILE called func2xyz is created for its compressed data.

It should be noted that text.arc can only be decompressed into separate files if -attributes=all is specified for the compression procedure, so that the names of all the original files are included in the FLAMFILE. Either of the output specifications [] or [*] can be used for the decompression procedure, to ensure that the files are decompressed separately and given their original names again. If an output file were to be specified, a single file would result, in which all the records of the original files would be concatenated.

2.5 Priority Rules for FLAM Settings

Settings such as the type of operation (compress or decompress), the input/output files, etc., can be entered either directly in the flam command or explicitly in a parameter file.

Implicit settings are also possible simply by omitting certain information, for example the compression mode for the compression procedure.

The settings which apply automatically if any specifications are missing or redundant are determined by evaluating certain information in a fixed sequence; this results in an order of priorities, which is based both on the category and on the origin of the information.

Since parameter files may refer to other parameter files, it is possible for several such files, possibly containing contradictory information, to be concatenated. The first of these parameter files has priority over all the others. If the same parameter is specified more than once within a parameter file, however, only the final specification is valid.
The information which is used to determine the FLAM settings is evaluated in the following order:

1. **The command itself**

2. **The parameter files**

3. **The installed default values**

If the command itself contains any contradictory or incompatible information, it is rejected and an error message is output.

The following priority rules apply in all other situations:

1. If a FLAM operation is specified in the command (compress/decompress), it has priority over all the other specifications. If the command does not contain such a specification, a search is made for it in the parameter file if one is named by -parfile=... or by default.

   If the operation is not specified as a FLAM parameter either, the installed default value, i.e. the default operation, is taken instead.

2. Any specifications which are incompatible with the FLAM operation are rejected.

3. All specifications missing in the command are set according to the parameter file, if one is specified either in the command itself or as an installation-specific default value.

4. All settings which cannot be established either at all or in part after rules 1 to 3 have been applied are determined on the basis of the installation-specific default values.

5. If the input or output specification is missing, the special stdin and stdout files are used.

In the case of FLAM parameters, specifications with a lower priority are always completely overridden by the higher priority, i.e. it is not possible to define a list of input specifications partly in the command itself and partly in the parameter file or in the form of default values. The list in the command cancels out the list at the lower specification level.

2.6 The FLAM parameter file

The parameter file can be edited using a text editor; it contains FLAM parameters which uniquely match the information in the command line of the flam command. Each FLAM parameter consists of a keyword and possibly an assigned value or a list of values. The FLAM parameters can be arranged so that each one is entered on a separate line; a line can however also contain several parameters separated by commas. Comments are allowed; they always start with a "!" and end at the line break.

The length of a line in the paramerter file must not exceed 2,048 bytes.

The example below illustrates how a parameter file can be used. FLAM is invoked by the following command:

```
FLAM -parfile=param.flam
```

The parameter file called param.flam contains the following FLAM parameters:

```
compress
mode=cx8
maxbuffer=128
flamin=test.dat
flamfile=flamfile1.cmp
attributes=all
recformat=fixed
recsize=1024
```

This FLAM call causes the file called test.dat to be compressed in the cx8 mode. The FLAMFILE is the file called flamfile1.cmp. It has a fixed record format and a record size of 1,024, and contains all the compression information, so that it can be decompressed in the same environment without having to specify the decompressed file.

The same FLAM parameters can also be arranged as follows:

```
compress,mode=cx8,maxbuffer=128
flamin=test.dat
flamfile=flamfile1.cmp
recformat=fix, recsize=1024
attributes=all
```

In this case, the output on the screen will be as follows:

```
FLAM (R) 4.2.0 for Linux
copyright (c) 2012 by limes datentechnik gmbh
Start: 06.08.2006  15:35:07

--------
compress
--------
mode ............................. adc
maxbuffer ........................ 65,536
maxrecords ....................... 4,096
flamcode ......................... ASCII
attributes ....................... all

Name of original file:............ /path001/test.dat
             inrecformat ......... stream
             inrecdelim .......... x'0a'

Number of uncompressed records ... 778
Number of uncompressed bytes ..... 26,947

Name of compressed file: ......... flamfile1.cmp
             recformat ........... fix
             recsize ............. 1,024

Number of compressed records ..... 14
Number of compressed bytes ....... 7,168

Compression efficiency (%) ....... 73.4
CPU time used (sec.) ............. 0.0
```

2.7 The FLAM Default Values File

The default values for the flam command are saved in the default values file flam_def.dat located in the lib subdirectory of the FLAM installation directory.

This file is given the following access rights: -rw-r--r--. It can be modified at any time by the system manager, but not **delete**d.

The default value file can be modified using the flam command with the -defaults parameter.

The settings in the default value file can be displayed using the flam –list command. No special privileges are necessary to do so.

When FLAM is invoked, any setting not provided in the command line nor via a parameter file is derived, if necessary, from the default values file which contains the installation-specific default values.

This is a binary file and may not be edited wit programs other than flam.

2.8 Alternative parameter names

Sometimes old parameters are replaced by new ones, in order to make the most of FLAM's enhanced functionality, while in other cases different, system-specific names are used to designate the same functions. Both the old names and those used by other systems are allowed in addition to the parameter names specified here, to ensure compatibility with Version 2.0 of FLAM for UNIX and with these other systems. However, it is always best to use the standard parameter names where possible.

Parameter name	Alternative name
-attributes = no	header=no
-attributes = common	-
-attributes = all	fileinfo, header=yes
decompress	uncompress
-flamcode	character_set
inrecdelim	in_format=recdelim irecdelim irdelim indelim
inrecformat	in_format informat irformat irecformat
inrecsize	in_format=(record-format) insize irsize irecsize
maxbuffer	buffer_size
maxrecords	records_in_buffer
msgfile	messages message_file log

Parameter name	Alternative name
mode=cx7	cx7 seven-bit
mode=cx8	cx8 eight-bit
mode=vr8	vr8
mode=adc	adc
option=cut	cut
option=nocut	nocut
outrecdelim	out_format=recdelim orecdelim ordelim outdelim
outrecformat	out_format orecformat orformat outformat
outrecsize	out_format=(record-format) orecsize orsize outsize
parfile	parameter_file
show=no show =error show =attributes show =all	info=no - info=hold info=yes
translate	code_table

FLAM (UNIX)

User Manual

Chapter 3:

The flamup Subprogram Call

3. Calling flamup

3.1 The flamup Calling Sequence

The application programs invoke the flamup subprogram in order to compress or decompress one or more files in accordance with the transferred arguments.

This section describes the data type and the semantics of the arguments. In real applications, the call must be adapted to the syntax of the respective programming language. In C, the arguments follow the name flamup; they are enclosed in parentheses and separated by commas.

flamup

The flamup interface provides application programs with the functionality of the flam command. User I/O and user exits are supported in addition.

Syntax

void **flamup** (char ***flamid*,
unsigned long **returncode*;
char **parameter_string*;
long **string_length*);

Arguments

flamid Specifies a long integer (4 bytes), which is used internally by FLAM during execution to identify the file.

returncode Specifies a long integer (4 bytes), which is used by flamup to return a return code to the invoking program.

parameter_string Specifies a string containing the FLAM parameters which must be used (see section 3.2, FLAM parameters in the flamup subprogram call). The individual FLAM parameters are separated by commas.

string_length Specifies a long integer (4 bytes) containing the length of the *parameter_string* argument in bytes. Maximal length is 512 bytes.

Return Codes

A description of the return codes can be found in Appendix A, Return Codes.

Application programs can convert the return codes which are returned by flamup to error messages, in accordance with the meanings listed in Appendix A, Return Codes. The internal message routine of the flam command can be used alternatively if desired with the FLAM table, instead of a separate table with error texts.

The call to output an error message with the tables used internally by FLAM is:

put_flmsg(return_code)

The definitions used by FLAM - in particular, those of the symbolic names for the various return codes - are contained in the C header file: /**usr**/**include**/**flamincl.h**

Application programs which are written in other programming languages require these definitions to be translated to the appropriate language by the programmer.

3.2 FLAM Parameters in the flamup Subprogram Call

The parameter_string argument of the flamup subprogram call may contain all the parameters which are valid for the flam command, with any of their valid values. The special rule which applies when lists of file names are specified should be noted. The individual file names must be separated by a comma "," and enclosed in parentheses "(",")" for the parameter concerned.

Example:

FLAM...=(file1,file2,..filen)

In addition to the parameters which are valid for the flam command, the subprogram call may contain the FLAM parameters for the user input/output routines (see also chapter 5) and the user exits (see also chapter 6). The following user input/output routines can be specified:

user_io

i(n)user_io

o(ut)user_io

The following user exits for file accesses can be specified:

exk10

exk20

exd10

exd20

The user input/output routines and the user exits must be generated by the user and linked to the application program and flamup (flam_upu.o module).

Only the parameters for the user input/output routines and user exits which are valid in flamup are described below.

exd10

exd10 specifies the name of a function which can be used to process the records of the decompressed file before they are written.

Syntax

exd10=*exit specification*

Values

exit specification

In this version, only exd10 may be specified for *exit specification*.

Description

exd10 defines a function which processes the records of the decompressed file(s) before they are written. This code must be generated by the user and linked to flamup and the embedding user program (see chapter 6).

exd20

exd20 specifies the name of a function which can be used to process the records of the compressed file after they are read.

Syntax

exd20=*exit specification*

Values

exit specification

In this version, only exd20 may be specified for *exit specification*.

Description

exd20 defines a function which processes the records of the compressed file(s) after they are read. This code must be generated by the user and linked to flamup and the embedding user program (see chapter 6).

exk10

exk10 specifies the name of a function which can be used to process the records of the original file after they are read.

Syntax

exk10=*exit specification*

Values

exit specification

In this version, only exk10 may be specified for *exit specification*.

Description

exk10 defines a function which processes the records of the original file(s) after they are read. This code must be generated by the user and linked to flamup and the embedding user program (see chapter 6).

exk20

exk20 specifies the name of a function which can be used to process the records of the compressed file before they are written.

Syntax

exk20=*exit specification*

Values

exit specification

In this version, only exk20 may be specified for *exit specification*.

Description

exk20 defines a function which processes the records of the compressed file(s) before they are written. This code must be generated by the user and linked to flamup and the embedding user program (see chapter 6).

inuser_io

This parameter specifies that the user-defined input/output routines should be used for the original file instead of those provided by FLAM.

Syntax inuser_io

Values None

Description The original file(s) are read with the input/output routines generated by the user. The usropn, usrget and usrcls functions are required to do so; the usrput and usrpos functions may be dummy functions (see chapter 5).

outuser_io

This parameter specifies that the user-defined input/output routines should be used for the decompressed file instead of those provided by FLAM.

Syntax outuser_io

Values None

Description The decompressed file(s) are read with the input/output routines generated by the user. The usropn, usrput and usrcls functions are required to do so; the usrget and usrpos functions may be dummy functions (see chapter 5).

user_io

This parameter specifies that the user-defined input/output routines should be used for the compressed file instead of those provided by FLAM.

Syntax

user_io

Values

None

Description

The compressed file(s) are read or written with the input/output routines generated by the user. The usropn and usrcls functions are always required to do so; usrput is required additionally for the compression procedure and usrget for the decompression procedure. usrpos is a dummy function. The function which is not required (usrget or usrput) may also be a dummy function (see chapter 5).

3.3 Linking flamup

flamup is installed in two prelinked modules, namely flam_up.o and flam_upu.o. The flam_up.o module contains dummy functions for the user input/output routines and the user exits. The flam_upu.o module contains no user input/output routines and no user exits.

flamup must be linked to the application program (xyz) with the following command:

```
ld -o xyz /opt/limes/flam/lib/flame/flam_up.o
xyz.c -lc
```

If the user wishes to use his own exits and/or his own input/output routines, they must be linked to the application program. In this case, the flam_upu.o module must be used instead of flam_up.o.

It should be noted that all the user exits and/or input/output routines must be generated by the user, and not just certain ones. If the user wants to use just his own exits or just his own input/output routines, he can link the dummy routines flam_usrmod.o and flam_exitmod.o, which are installed in /opt/limes/flam/lib/flame, for the other functions.

In this case, the command is as follows:

```
ld -o xyz /opt/limes/flam/lib/flame/flam_upu.o
/opt/limes/flam/lib/flame/flam_usrmod.o<D>
/opt/limes/flam/lib/flame/flam_exitmod.o xyz.c -lc
```

/opt/limes/flam/lib/flame/flam_usrmod.o and/or /opt/limes/flam/lib/flame/flam_exitmod.o must be replaced by the user's own modules.

Some UNIX systems require the runtime library /lib/crt0.o to be specified in the link command as well, e.g. SCO UNIX.

FLAM (UNIX)

User Manual

Chapter 4:

The Record Interface flamrec

4. The Record Interface flamrec

4.1 Functions of the Record Interface

The record interface, flamrec, consists of a series of subprograms, which can be invoked by all programming languages in which C functions can be used. With the exception of the key description, which is used in a later version, all arguments are represented using elementary data types (long integer, string). The arguments are transferred by means of pointers and not as values. The rules concerning notation in section 2.5, Interfaces apply here analogously.

Application programs which are written in C may include the statement #include "flamincl.h". This C header file contains the definitions of the symbolic constants and return codes, as well as the structure of the key description. These definitions must be transferred in the same way for programs written in other languages.

All argument lists begin with an ID, which identifies the compression file uniquely. This flmid argument contains the address of the work area for the compressed file, which was specified by flmopn, and must not be modified until flmcls. All other arguments are only relevant to the function to which they are transferred.

The returncode argument is likewise contained in every argument list; it serves either to confirm successful execution of a function or to report an error, if one has occurred. All the codes and their meanings are listed in Appendix A, Return Codes, together with the meaning of the additional information supplied in the most significant byte of the return code for all error types which occur in connection with file accesses. As with the flamup subprogram, the return codes of the record interface functions can be converted to message numbers or output with FLAM's internal message routine put_flmsg.

The record interface comprises the following functions:

Function	Purpose	Corresponding I/O calls
flmcls	Terminate processing of original file and close FLAMFILE	close
flmflu	Terminate processing of original file without closing FLAMFILE	fflush
flmget	Reading original record sequentially	(f)gets/ read
flamghd	Reading common FLAM file header	
flmguh	Reading user-specific FLAM file header	
Flmopn, flmopd, flmopf	Opening a FLAMFILE	create/ (f)open
flmphd	Writing common FLAM file header	
flmpos	Positioning to next FLAM file header	
flmpuh	Writing user-sepcific FLAM file header	
flmput	Writing original record sequentially	(f)puts/ write
flmpwd	Passing a password	

4.2 Programming the Record Interface at Compression

The process of creating a FLAMFILE can be divided into three phases:

- open sequence

- one or more compression cycles

- termination (function **flmcls**)

The open sequence always starts calling the **flmopn** function. Subsequently, **flmopd** and/or **flmopf** can be called if **flmopn** was called with continue_param set. When both functions are called, **flmopd** must come first and also have continue_param set.

When **flmopd** or **flmopf** are not called by the application program, FLAM uses implicit settings to generate such calls internally. Default values used with the flam command and flamup calls are not effective here.

Finally, if encryption is used, a password must be provided by a **flmpwd** call.

In the second phase, one compression cycle must be done for each file being compressed.

Such a cycle would normally begin with a **flmphd** call to generate a common FLAM fileheader. Only when a single file is compressed without encryption, this may be omitted.

If a FLAM fileheader was created, a user-specific header may be appended by invoking **flmpuh**. With Secure FLAMFILEs, this invocation is mandatory even if the header data length is 0.

Now data can be submitted for compression by repeated execution of **flmput**. Each such call passes a record in the FLAM terminology, which can be later replicated identically or with modified record attributes thanks to the structure information kept by FLAM. FLAM collects the data passed to it in the compression buffer and controls compression and the resulting output without involving the application program.

A compression cycle is terminated by a call to **flmflu** which causes the remainder in the compression buffer to be compressed and the output of the result. This function also returns statistical information to the application.

Following this, a new compression cycle may be initiated by another **flmphd** or the FLAMFILE can be closed by calling **flmcls**.

Control is always returned to the invoking program. There are no error exits and no error messages are generated by the record interface. Rather, a return code is passed back to the application.

4.3 Programming the Record Interface at Decompression

As with compression, decompressing a FLAMFILE also consists of three phases:

- open sequence

- decompression cycle(s)

- termination (function **flmcls**)

The open sequence here consists of the same function calls as at compression. Only the direction of the flow of information is reversed for some of the arguments, e.g. the compression mode. They are returned to the application.

Every function used in the compression cycle has its decompression counterpart that performs the complementary operation. The complements for **flmphd** and **flmpuh** are **flmghd** and **flmguh**, respectively, that for **flmput** is **flmget**. A decompressin cycle is also terminated with **flmflu**. In addition, there is a function, **flmpos**, which advances the read position in a FLAMFILE to the beginning of the next original file. This function has no counterpart with compression.

Typically, a decompression cycle initially invokes **flmghd** which returns details about the original file such as file name, record format, etc. This may be followed by **flmguh** to retrieve the user-specific header, if present. Subsequent **flmget** calls retrieve one record per call, with the last record of an original file being signalled by a special return code. Skipping the remaining records of an original file can be done by **flmpos** which positions to the next file header. A **flmflu** terminates a decompression cycle that may be followed by another cycle or by **flmcls**, which terminates the processing. One **flmflu** or **flmpos** per cycle is mandatory, all other calls are optional.

4.4 Description of flamrec Functions

The next section describes the functions of the record interface in alphabetical order.

flmcls

The flmcls (close) function terminates the access to the record interface. After the final matrix has been compressed during the compression procedure, the compressed data is written in the FLAMFILE and the FLAMFILE is closed. With Secure FLAMFILEs, a file trailer is appended.

During the decompression procedure, only the FLAMFILE is closed; any remaining original records are not transferred.

Statistical information is returned as well if it has been requested with flmopn (*statistics* ≠ 0).

Syntax

```
void flmcls ( char **flmid,
              long *returncode,
              unsigned long *cputime,
              unsigned long *records,
              unsigned long *bytes,
              unsigned long *byteofl,
              unsigned long *cmprecs,
              unsigned long *cmpbytes,
              unsigned long *cmpbytofl);
```

Arguments

flmid — The contents of *flmid* is a pointer to a character string. This argument identifies the FLAMFILE for which the function is being executed. It is set by the flmopn function and must not be changed until flmcls is done.

returncode — This argument returns a return code to the invoking program.

cputime — This argument returns the current time in seconds units.

records — If statistical information has been requested with flmopn (statistics ≠ 0), this argument returns the number of decompressed records which have been transferred with flmput or flmget. The accumulated values of this counter are only significant if complete files are processed.

bytes — If statistical information has been requested with flmopn (statistics ≠ 0), this argument returns the number of bytes in the decompressed records which have been transferred with flmput or flmget. The accumulated values of this counter are only significant if complete files are processed.

byteofl — If the accumulated value of the bytes argument exceeds 2,000,000,000, this counter is used for multiples of 2,000,000,000.

cmprecs If statistical information has been requested with flmopn (statistics ≠ 0), this argument returns the number of compressed records which were created during the compression procedure or read during the decompression procedure. This value is only significant if complete files are processed.

cmpbytes If statistical information has been requested with flmopn (statistics ≠ 0), this argument returns the number of bytes which were created during the compression procedure or read during the decompression procedure. This value is only significant if complete files are processed.

cmpbytofl If the accumulated value of the cmpbytes argument exceeds 2,000,000,000, this counter is used for multiples of 2,000,000,000.

Dependencies

With Secure FLAMFILEs, flmcls may only be invoked immediately after flmflu or flmpos.

Return codes

A description of the return codes can be found in Appendix A, Return codes.

flmflu

flmflu (flush) terminates the compression procedure for a file; the remaining records in the block are compressed and written with padding of the last FLAM record, where necessary. The compressed file is not closed after flmflu and the statistics counters are not reset. With Secure FLAMFILEs, a member trailer is appended.

Syntax

```
void flmflu ( char **flmid,
              long *returncode,
              unsigned long *cputime,
              unsigned long *records,
              unsigned long *bytes,
              unsigned long *byteofl,
              unsigned long *cmprecs,
              unsigned long *cmpbytes,
              unsigned long *cmpbytofl);
```

Arguments

flmid — The contents of *flmid* is a pointer to a character string. This argument identifies the FLAMFILE for which the function is being executed. It is set by the flmopn function and must not be changed until flmcls is done.

returncode — This argument returns a return code to the invoking program.

cputime — This argument returns the current time in seconds units.

records — If statistical information has been requested with flmopn (statistics ≠ 0), this argument returns the number of decompressed records which have been transferred with flmput or flmget. The accumulated values of this counter are only significant if complete files are processed.

bytes — If statistical information has been requested with flmopn (statistics ≠ 0), this argument returns the number of bytes in the decompressed records which have been transferred with flmput or flmget. The accumulated values of this counter are only significant if complete files are processed.

byteofl — If the accumulated value of the bytes argument exceeds 2,000,000,000, this counter is used for multiples of 2,000,000,000.

cmprecs — If statistical information has been requested with flmopn (statistics ≠ 0), this argument returns the number of compressed records which were created during the compression procedure or read during the decompression procedure. This value is only significant if complete files are processed.

cmpbytes If statistical information has been requested with flmopn (statistics ≠ 0), this argument returns the number of bytes which were created during the compression procedure or read during the decompression procedure. This value is only significant if complete files are processed.

cmpbytofl If the accumulated value of the cmpbytes argument exceeds 2,000,000,000, this counter is used for multiples of 2,000,000,000.

Dependencies When compressing Secure FLAMFILEs, flmflu is permitted only after flmpuh or flmput. When decompressing Secure FLAMFILEs, flmflu must be preceded either by flmget, flmghd, or flmguh.

Return codes A description of the return codes can be found in Appendix A, Return codes.

flmget

The flmget (get record) function reads the next original record sequentially. The data is transferred to the record buffer of the invoking program. A special return code indicates when a new file header is encountered.

Syntax

void **flmget** (char ***flmid*,
long **returncode*,
long **record_length,*
char **record*,
long **buffer_length*);

Arguments

flmid The contents of *flmid* is a pointer to a character string. This argument identifies the FLAMFILE for which the function is being executed. It is set by the flmopn function and must not be changed until flmcls is done.

returncode This argument returns a return code to the invoking program.

record_length This argument returns the length of the record which has been read in bytes.

record This argument is the record buffer.

buffer_length This argument contains the length of the record buffer in bytes.

Dependencies

flmget may only be used at decompression. With encrypted FLAMFILEs, a password must be provided via flmpwd before the first invocation of flmget.

Return codes

A description of the return codes can be found in Appendix A, Return codes.

flmghd

flmghd (get file header) allows retrieving the file attributes of the original file during decompression if they were stored in the FLAMFILE during compression.

If the FLAMFILE contains several file headers (see flmphd), the last file header found by FLAM is transferred with flmghd.

Syntax

```
void flmghd (char **flmid,
             long *returncode,
             long *filename_length,
             char *filename,
             long *organization,
             long *record_format,
             long *record_size,
             char *record_delimiter,
             struct kd *key_description,
             long *block_size,
             long *print_control,
             char *system);
```

Arguments

flmid — The contents of *flmid* is a pointer to a character string. This argument identifies the FLAMFILE for which the function is being executed. It is set by the flmopn function and must not be changed until flmcls is done.

returncode — This argument returns a return code to the invoking program.

filename_length — This argument contains the length of the buffer area in which the name of the original file must be returned. The actual length is also returned by it.

filename — This argument specifies the buffer area in which the name of the original file must be returned. The name is saved there in the form of a character string. If the area is not long enough, only the number of characters in the name which the area can actually hold are returned.

organization

This argument returns a value for the organization of the original file. The possible values and their meanings, some of which refer to organization types which either do not exist at all in UNIX or which are not supported by it, are as follows:

0	sequential
1	index-sequential
2	relative
3	direct access
4	no record structure
5	library
6	physical

record_format

This argument returns a value for the record format of the original file. The possible values and their meanings, some of which refer to record formats which do not exist in UNIX, are as follows:

0	variable
1	fixed
2	undefined
3	stream
8	var/blocked
9	fix/blocked
16	var/blocked/spanned
17	fix/blocked/standard
18	eaf
24	vfc
32	var_4b
40	var_ascii
48	var_ebcdic

record_size

This argument returns the record size of the original file. The value 0 is set for variable record sizes and for the stream record format.

record_delimiter

This argument specifies the record delimiter for the stream record format. The character string is 4 bytes long (corresponding to a long integer).

The record_delimiter has the following structure:

The most significant byte contains the number of bytes. If the number of bytes is 1, the least significant byte contains the record delimiter. If the number of bytes is 2, the second least significant byte contains the first byte of the record delimiter and the least significant byte contains the second byte of this delimiter, e.g.:

0x0100000a	record delimiter LF (0x0a)
0x02000d0a	record delimiter CR/LF (0x0d0a)

key_description This argument is reserved for future extensions.

block_size This argument is reserved for future extensions.

print_control This argument is reserved for future extensions.

system This argument consists of two bytes and is used to return a code for the operating system in which the FLAMFILE was created. The meaning of this hexadecimal code is as follows:

00 00 Unknown system
00 0E MS-Windows
01 01 IBM MVS
01 02 IBM VSE
01 03 IBM VM
03 01 HP OpenVMS
09 01 HP NonStop
12 04 Unix

Dependencies

flmghd may only be used at decompression. It may be called to retrieve the attributes of the first member of a FLAMFILE, is FLAM returns code 1 after the opening sequence (flmopn / flmopd / flmopf). The attributes of a subsequent member are available if a preceding flmget returns with code 6.

Return codes

A description of the return codes can be found in Appendix A, Return codes.

flmguh

The flmguh (get user header) function is used to recover the data from the user-specific FLAM file header during the decompression procedure. It is only allowed during this procedure and immediately following an flmghd call.

Syntax

void **flmguh** (char ***flmid*,
long **returncode*,
long **user_attributes_length*,
char **user_attributes*);

Arguments

flmid The contents of *flmid* is a pointer to a character string. This argument identifies the FLAMFILE for which the function is being executed. It is set by the flmopn function and must not be changed until flmcls is done.

returncode This argument returns a return code to the invoking program.

user_attributes_length This argument transfers the length of the memory area in in which the user-specific FLAM file header must be saved. It returns the length which is actually used.

user_attributes This argument returns the user-specific FLAM file header.

Dependencies

flmguh may only be used at decompression. It returns data from the user-specific header that were stored with flmpuh.

Return codes

A description of the return codes can be found in Appendix A, Return codes.

flmopd

The flmopd (open/file description) function enables special attributes of the FLAMFILE to be set or interrogated explicitly. This call is not necessarily mandatory. If it is not executed during the compression procedure, implicit values are used instead for the relevant attributes of the FLAMFILE. These are specified, where applicable, together with the descriptions of the various arguments.

flmopd can only be invoked following flmopn.

Syntax

```
void flmopd (char **flmid,
             long *returncode,
             long *continue_param,
             long *filename_length,
             char *filename,
             long *organization,
             long *record_format,
             long *record_size,
             char *record_delimiter,
             struct kd *key_description,
             long *block_size,
             long *close_disposition,
             long *device);
```

Arguments

flmid
The contents of flmid is a pointer to a character string. This argument identifies the FLAMFILE for which the function is being executed. It is set by the flmopn function and must not be changed until flmcls is done.

returncode
This argument returns a return code to the invoking program.

continue_param
This argument indicates whether or not more setting data should subsequently be transferred with an flmopf call.

0	No flmopf call
Other values	flmopf call follows

filename_length
This argument transfers the length of the area for the name of the FLAMFILE which must be opened and returns the actual length of this name.

filename
This argument is the area in which the name of the FLAMFILE which must be opened is returned.

organization This argument transfers and returns a code for the organization of the FLAMFILE which must be opened.

The possible values are:

0	sequential

Implicit value (at compression): 0.

record_format This argument transfers and returns a code for the record format of the FLAMFILE which must be opened.

Valid values are:

0	var with mode=adc/cx8/vr8
1	fixed with mode=adc/cx8/vr8
3	stream only with mode=cx7

Implicit value (at compression): 1.

record_size This argument passes and returns the maximum record size of the FLAMFILE to be opened.

Implicit value (at compression): 512.

record_delimiter This character string is 4 bytes long (corresponding to a long integer).

The structure of the record_delimiter is described under the flmghd function.

key_description The argument is reserved for future extensions.

block_size This argument is reserved for future extensions.

close_disposition This argument is reserved for future extensions.

device This argument is used to activate the user-defined input/output. These routines must be made available instead of the input/output instructions and linked to the application program.

7	User input/output
Other values	Standard

Implicit value (at compression): 0.

Dependencies

flmopd is only permitted after a call to flmopn with *continue_param* ≠ 0. The flmopd call itself must have *continue_param* ≠ 0, if flmopf is to be called subsequently.

When called with *device=7*, the application program must be linked with user-I/O routines as described in section 5.2.

Return codes

A description of the return codes can be found in Appendix A, Return codes.

flmopf

The flmopf (open FLAM settings) function enables the FLAM settings to be set or interrogated explicitly.This call is not necessarily mandatory. If it is not executed during the compression procedure, implicit values are used instead for the relevant settings. These are specified, where applicable, together with the descriptions of the various arguments.

flmopf can only be used as the final function call of an open sequence, in other words only after flmopd if this is invoked or only after flmopn if it is not.

Syntax

```
void flmopf ( char **flmid,
              long *returncode,
              long *flam_version,
              long *flam_code,
              long *comp_mode,
              long *max_buffer,
              long *fileheader,
              long *max_records,
              struct kd *key_description,
              long *block_mode,
              char *exitroutine_comp,
              char *exitroutine_decomp);
```

Arguments

flmid The contents of *flmid* is a pointer to a character string. This argument identifies the FLAMFILE for which the function is being executed. It is set by the flmopn function and must not be changed until flmcls is done.

returncode This argument returns a return code to the invoking program.

flam_version The value of this argument must always be 2.

flam_code During compression, this argument specifies the character set which must be used to write the character-coded information in the FLAMFILE. This information is returned during the decompression procedure.

0 EBCDIC
1 ASCII

Implicit value (at compression): 1.

comp_mode

During compression, this argument specifies which compression and/or encryption method to apply. This information is returned at decompression.

The following hexadecimal (decimal) values are permitted:

Compression method	Encryption method		
	none	**FLAMenc**	**AES**
cx8	0x00 (0)	not permitted	not permitted
cx7	0x01 (1)	not permitted	not permitted
vr8	0x02 (2)	not permitted	not permitted
adc	0x03 (3)	0x13 (19)	0x23 (35)
ndc	0x0B (11)	0x1B (27)	0x2B (43)

Implicit value (at compression): 0.

max_buffer

During compression, this argument specifies the maximum buffer size which is to be used. This information is returned as a number of bytes during decompression.

The buffer size is specified in either bytes or Kbytes. All values between 1 and 2,621,440 are valid. Details on how this value is interpreted and the buffer sizes actually used are found in the description of the maxbuffer parameter in section 2.3.

Implicit value (at compression): 32 Kbytes.

fileheader

During decompression, the information returned by this argument indicates whether or not the FLAMFILE contains a FLAM file header. This determines whether or not a subsequent flmghd call is meaningful.

The argument is not used during the compression.

0 No FLAM file header
1 FLAM file header

Implicit value (at compression): 0.

max_records

During compression, this argument specifies the maximum number of records which are to be used per matrix. This information is returned during decompression.

Implicit value (at compression): 255.

key_description This argument is reserved for future extensions.

block_mode This argument is reserved for future extensions.

exitroutine_comp This argument contains the name of a user exit, in other words an executable program which is invoked each time the FLAMFILE is accessed during compression. This program must be linked to the application program as a C function with the name exk20. The user exit is not invoked if the name begins either with a blank (X'20') or with a binary zero (X'00').

If the user exit exitroutine_comp modifies the compressed record, a corresponding user exit *exitroutine_decomp* must be specified for decompression, so that the changes are undone again before the record is processed by FLAM.

exitroutine_decomp This argument contains the name of a user exit, in other words an executable program which is invoked each time the FLAMFILE is accessed during decompression. This program must be linked to the application program as a C function with the name exd20.

The user exit is not invoked if the name begins either with a blank (X'20') or with a binary zero (X'00').

A user exit *exitroutine_decomp* must be specified if FLAMFILE records have been modified during the compression procedure by a user exit exitroutine_comp and all the changes need to be undone again by this exit.

Dependencies

flmopf may only be invoked immediately after a call to flmopn or flmopd with *continue_param* ≠ 0.

Return codes

A description of the return codes can be found in Appendix A, Return codes.

flmopn

The flmopn (open) function starts the open sequence for a FLAMFILE. It can be followed by a flmopd and/or flmopf call, in order to set or query the attributes of the FLAMFILE or the FLAM settings.

Syntax

```
void flmopn (char **flmid;
             long *returncode;
             long *continue_param;
             long *flam_open;
             char *filename;
             long *statistics);
```

Arguments

flmid
The contents of *flmid* is a pointer to a character string. This argument identifies the FLAMFILE for which the function is being executed. It is set by the flmopn function and must not be changed until flmcls is done.

returncode
This argument returns a return code to the invoking program.

continue_param
This argument indicates whether or not more setting data should subsequently be transferred with an flmopd or flmopf call.

0	No more calls
Other values	More calls follow

flam_open
This argument specifies the FLAMFILE processing mode.

0	input (read FLAMFILE, i.e. decompress)
1	output (write FLAMFILE, i.e. compress)

filename
This argument transfers the name of the FLAMFILE. The string must be terminated with a zero byte (binary zero). The file name is returned by the flmopd function.
If stdin/stdout are used, the first byte contains either a binary zero or a blank.

statistics
This argument specifies whether or not statistical information is desired at the end of the compression procedure.

0	No statistics
Other values	Statistical information returned

Dependencies

None.

Return codes

A description of the return codes can be found in Appendix A, Return codes.

flmphd

The flmphd (put file header) function is only allowed for the compression procedure. It generates a common file header, which describes the file format of the original records that are transferred subsequently. If several different files are compressed into a single FLAMFILE, a separate file header can be generated for each file with the flmphd function.

FLAM returns this file header information during the decompression procedure if it has been requested to do so (with flmghd).

Syntax

```
void flmphd (char **flmid,
             long *returncode,
             long *filename_length,
             char *filename,
             long *organization,
             long *record_format,
             long *record_size,
             char *record_delimiter,
             struct kd *key_description,
             long *block_size,
             long *print_control,
             char *system,
             long *continue_param);
```

Arguments

flmid — The contents of *flmid* is a pointer to a character string. This argument identifies the FLAMFILE for which the function is being executed. It is set by the flmopn function and must not be changed until flmcls is done.

returncode — This argument returns a return code to the invoking program.

filename_length — This argument contains the length of the buffer area in which the name of the original file is transferred.

filename — This argument specifies the buffer area in which the name of the original file is transferred. The name is saved there as a string and terminated with '\0'.

organization — This argument transfers a code for the organization of the original file.

The permissible values are described under the flmghd function.

record_format This argument transfers a code for the record format of the original file.

The permissible values are described under the flmghd function.

record_size This argument transfers the maximum record size of the original file. The permissible values are described under the flmghd function.

record_delimiter The record delimiter is saved in this argument if the record format has been specified as stream. The character string is 4 bytes long (corresponding to a long integer).

The structure of the record_delimiter is described under the flmghd function.

key_description This argument is reserved for future extensions.

print_control This argument is reserved for future extensions.

system This argument consists of two bytes and is used to transfer a code for the operating system in which the FLAMFILE was created.

The list of valid values is listed with flmghd.

continue_param This argument indicates whether or not more information for the FLAM file header should subsequently be transferred with an flmpuh call.

0	No more information
Other values	More information

Dependencies flmphd may only be used at compression and only if flmopf was called before with *fileheader*=1. Calls to flmphd with *continue_param* ≠ 0 or after flmpwd must be followed immediately by flmpuh.

Return codes A description of the return codes can be found in Appendix A, Return codes.

flmpos

The flmpos (position) function can be used to advance to the end of the data in an original file during the decompression procedure, in order to read the file header of the next file.

Syntax

void **flmpos** (char ***flmid*;
long **returncode*;
long **position*);

Arguments

flmid The contents of *flmid* is a pointer to a character string. This argument identifies the FLAMFILE for which the function is being executed. It is set by the flmopn function and must not be changed until flmcls is done.

returncode This argument returns a return code to the invoking program.

position This argument effects a jump to the end of the data in an original file during a decompression procedure with 99,999,998, in other words either to the next FLAM file header or to the end of the FLAMFILE.

Dependencies

flmpos may only be used at decompression.

Return codes

A description of the return codes can be found in Appendix A, Return codes.

flmpuh

The flmpuh (put user header) function enables additional information with any structure to be saved in the FLAMFILE. This information is inserted in the form of a string and can be made available during the decompression procedure with flmguh (the complementary function).

The flmpuh function is only allowed during the compression procedure and immediately following an flmphd call.

Syntax

void **flmpuh** (char ***flmid*,
long **returncode*,
long **user_attr_length*,
char **user_attributes*);

Arguments

flmid The contents of *flmid* is a pointer to a character string. This argument identifies the FLAMFILE for which the function is being executed. It is set by the flmopn function and must not be changed until flmcls is done.

returncode This argument returns a return code to the invoking program.

user_attr_length This argument passes the length of the user-specific header (maximum length 32732 bytes).

user_attributes This argument specifies a memory area containing the user-specific header, the contents of which can be freely defined by the user.

Dependencies

flmpuh may only be called at compression. It may be used only immediately after flmphd and is mandatory when a Secure FLAMFILE is being created.

Return codes

A description of the return codes can be found in Appendix A, Return codes.

flmput

The flmput (put record) function transfers one original record at a time for compression.

Syntax

```
void flmput (flmid, returncode, record_length, record)
            char **flmid,
            long *returncode,
            long *record_length,
            char *record);
```

Description

flmid The contents of *flmid* is a pointer to a character string. This argument identifies the FLAMFILE for which the function is being executed. It is set by the flmopn function and must not be changed until flmcls is done.

returncode This argument returns a return code to the invoking program.

record_length This argument contains the record length in bytes.

record This argument is the record buffer.

Dependencies

flmput may only be used at compression.

Return codes

A description of the return codes can be found in Appendix A, Return codes.

flmpwd

Mit der Funktion flmpwd (password) wird beim Komprimieren ein Kennwort zum Ver-, beim Dekomprimieren zum Entschlüsseln der Komprimatsdaten übergeben.

Syntax

void **flmpwd** (char ***flmid*,
long **returncode*,
long **pwd_length*,
char **pwd*);

Arguments

flmid The contents of *flmid* is a pointer to a character string. This argument identifies the FLAMFILE for which the function is being executed. It is set by the flmopn function and must not be changed until flmcls is done.

returncode This argument returns a return code to the invoking program.

pwd_length This argument contains the password's length in characters.

pwd This argument is the buffer containing the password string.

Dependencies

flmpwd may only be used when compression or decompression is done with encryption/decryption.

Return codes

A description of the return codes can be found in Appendix A, Return codes.

4.5 Linking Record Interface Functions to Application Programs

FLAM's record interface is installed in two prelinked modules, namely flam_rec.o and flam_recu.o. The flam_rec.o module contains the dummy functions for the user input/output and the user exits. The flam_recu.o module contains no user input/output routines and no user exits.

The record interface must be linked to the user program (xyz) with the following command:

```
ld -o xyz /opt/limes/flam/lib/flame/flam_rec.o
xyz.c -lc
```

If the user wishes to use his own exits and/or his own input/output routines, they must be linked to the application program. In this case, the flam_recu.o module must be used instead of flam_rec.o.

It should be noted that all the user exits and/or input/output routines must be generated by the user, and not just certain ones.If the user wants to use just his own exits or just his own input/output routines, he can link the dummy routines flam_usrmod.o and flam_exitmod.o, which are installed in /opt/limes/flam/lib/flame, for the other functions.

In this case, the command is as follows:

```
ld -o xyz /opt/limes/flam/lib/flame/flam_recu.o
/opt/limes/flam/lib/flame/flam_usrmod.o <D>
/opt/limes/flam/lib/flame/flam_exitmod.o xyz.c -lc
```

/opt/limes/flam/lib/flame/flam_usrmod.o and/or /opt/limes/flam/lib/flame/flam_exitmod.o must be replaced by the user's own modules.

Some UNIX systems require the runtime library /lib/crt0.o to be specified in the link command as well.

FLAM (UNIX)

User Manual

Chapter 5:

The User Input/Output Interface

5. The User Input/Output Interface

5.1 How to Use the User Input/ Output Interface

FLAM uses the standard functions to input and output original files, decompressed files and FLAMFILEs. It is however also possible to process data in a non-supported format, or on hardware which is not supported by UNIX, with the aid of programs which invoke the FLAM functions of the record interface. The necessary input and output routines must be made available in order to do so. FLAM uses these routines for the file concerned if user input/output (device=7) is specified in the device argument when the FLAM flmopd function is invoked.

The most important difference between the user input/output interface and the record interface is that the user program is the invoking program and FLAM the executive program for the record interface functions. User inputs/outputs, on the other hand, are themselves executive routines and are invoked by FLAM. They thus have no control over the order in which they are activated and must respond context-sensitively to each call. If they need RAM in order to do so, the file is assigned an area of 1 Kbyte for each usropn call; the same area remains assigned each time this file is called subsequently.

Since the FLAM concept always entails reading data from one file and writing it in another, user input/output routines are simply alternative file accesses from FLAM's point of view, irrespective of the actions which are actually performed by them.

User input/output routines must therefore incorporate the following functions:

- usrcls
- usrget
- usropn
- usrpos
- usrput

These functions must conform to the interface conventions, i.e. they must be defined with the specified name, the type and length of their arguments must match and their return codes must reflect the success or error situation correctly.

Instructions for linking user input/output routines can be found at the end of this chapter.

usrcls

This function closes a file.

Syntax

void **usrcls** (char ***workio*,
long **returncode*);

Arguments

workio This argument is a 1 Kbyte area, which is made available by FLAM as a work area each time a user input/output function is invoked and which is not altered by FLAM between the calls.

returncode This argument returns a return code to the invoking program.

Return codes

A description of the return codes can be found in Appendix A, Return codes.

usrget

This function reads a record sequentially and transfers it to FLAM.

Syntax

```
void usrget (char **workio,
             long *returncode,
             long *record_length,
             char *record,
             long *buffer_length);
```

Arguments

workio — This argument is a 1 Kbyte area, which is made available by FLAM as a work area each time a user input/output function is invoked and which is not altered by FLAM between the calls.

returncode — This argument returns a return code to the invoking program.

record_length — This argument returns the length of the record which has been read in bytes.

record — This argument is the record buffer.

buffer_length — This argument contains the length of the record buffer in bytes.

Return codes

A description of the return codes can be found in Appendix A, Return codes.

usropn

This function opens the file whose specification is transferred by the filename argument.

Syntax

```
void usropn (char **workio,
             long *returncode,
             long *file_open,
             char *fname,
             long *organization,
             long *record_format,
             long *record_size,
             long *block_size,
             struct kd *key_description,
             long *device,
             char *record_delimiter,
             char *pad_char,
             long *print_control,
             long *close_disposition,
             long *access,
             long *filename_length,
             char *fname);
```

Arguments

workio

This argument is a 1 Kbyte area, which is made available by FLAM as a work area each time a user input/output function is invoked and which is not altered by FLAM between the calls.

When usropn is invoked, the name of the file which must be processed is at the start of the area and the remainder of the area is filled with binary zeros.

returncode

This argument returns a return code to the invoking program.

file_open

This argument specifies the processing mode of the file which must be opened.

0 input (read, i.e. decompress)

1 output (write, i.e. compress)

filename

This argument passes the specification for the file which must be opened. The string must be terminated with a zero byte (binary zero). The specification may also be a logical name. The actual file name should be returned by this argument

The contents of this argument are also located at the start of the memory area which was passed with the workarea argument.

organization

This argument is reserved for future extensions.

record_format This argument transfers and returns a code for the record format of the file which must be opened.

The values are defined as follows:

0	variable/var_2b
1	fixed
2	undefined
3	stream
18	eaf
32	var_4b
40	var_ascii
48	var_ebcdic

record_size This argument transfers the record size of the file and may return this value for an input file. The value 0 is set for variable record formats. A value other than 0 is required for the fixed and undefined record formats.

block_size This argument is reserved for future extensions.

key_description The argument is reserved for future extensions.

device This argument is reserved for future extensions.

record_delimiter This argument specifies the record delimiters which must be used when files with the stream record format are opened. If the argument contains the value 0, the default delimiters are used. If not, the record delimiter is the contents of this argument.

The record_delimiter has the following structure:

The most significant byte contains the number of bytes. If the number of bytes is 1, the least significant byte contains the record delimiter. If the number of bytes is 2, the second least significant byte contains the first byte of the record delimiter and the least significant byte contains the second byte of this delimiter, e.g.:

"0a 00 00 01"	UNIX record delimiter
"0a 0d 00 02"	MS-DOS record delimiter

pad_char This argument transfers a padding character, which is used to pad short fixed-size records when they are output.

print_control This argument is reserved for future extensions.

close_disposition	This argument is reserved for future extensions.
access	This argument is reserved for future extensions.
filename_length	This argument passes the length of the string (filename) of the file which must be opened and returns the length of the name.
fname	This argument is reserved for future extensions.

Return codes

A description of the return codes can be found in Appendix A, Return codes.

usrpos

The usrpos function increments the record key when a relative file is decompressed if gaps need to be generated in the decompressed file.

This function is not used (it is a dummy function).

Syntax

void **usrpos** (char ***workio*,
long **returncode*,
long **position*);

Arguments

workio This argument is a 1 Kbyte area, which is made available by FLAM as a work area each time a user input/output function is invoked and which is not altered by FLAM between the calls.

returncode This argument returns a return code to the invoking program.

position This argument transfers the number of record positions which must be skipped.

Return codes

A description of the return codes can be found in Appendix A, Return codes.

usrput

This function writes a record which has been transferred by FLAM sequentially.

Syntax

```
void usrput  (char **workio,
              long *returncode,
              long *record_length,
              char *record);
```

Arguments

workio — This argument is a 1 Kbyte area, which is made available by FLAM as a work area each time a user input/output function is invoked and which is not altered by FLAM between the calls.

returncode — This argument returns a return code to the invoking program.

record_length — This argument contains the record length in bytes.

record — This argument is the record buffer.

Return codes

A description of the return codes can be found in Appendix A, Return codes.

5.2 Linking User Input/Output Routines to Application Programs

Data access with the aid of user input/output routines is supported by FLAM both for subprogram interface calls and for record interface calls.

In order to be able to access data with FLAM by means of user input/output routines, the associated linked object files must be made available for all the access functions described above; any functions which are not required can be installed as dummy functions.

How to link the routines to the application program is described in section 3.3, "Linking flamup ", as well as in section 4.5, "Linking Record Interface Functions to Application Programs".

FLAM (UNIX)

User Manual

Chapter 6:

The User Exits

6. The User Exits

A user exit is a user-provided program that is activated by FLAM and which communicates with it through a defined interface.

FLAM supports two sorts of user exits:

- exits activated when files are accessed, i.e. opened, closed, read, or written, also called access exits;
- an exit for automatic key management, also called key exit.

In order to be available in FLAM, the access exits must be (statically) linked with FLAM (see section 6.1.2), whereas the key exit is created as a shared object and linked dynamically, i.e. at runtimes. For details see section 6.2.2.

6.1 User Exits for File Accesses (Access Exits)

These exits are programs which are activated by FLAM with each operation on the associated file. When the file is opened or accessed for reading, the user exit is activated immediately after the file has been opened or after a record has been transferred to the read buffer, and before FLAM begins processing.

When the file is closed or accessed for writing, the user exit is activated immediately before the closing or before the output takes place, and after FLAM has finished processing. The user exit can evaluate and manipulate the record concerned, or possibly insert additional records, before handing control back to FLAM. At the same time, FLAM expects the user exit to supply instructions in the form of a return code, telling it either how to continue processing or that it should abort due to an error.

User exits can be specified for FLAMFILEs when calls are made via the subprogram or record interface. If FLAM calls are made using the record interface functions, only user exits for the FLAMFILE can be activated, since FLAM is not able to read or write the original and decompressed files. For FLAM calls via the subprogram interface flamup, access exits can be specified both for FLAMFILEs as well as for original or decompressed files. The record interface flamrec only allows activating exits for FLAMFILEs since access to original or decompressed files then lies with the application program.

FLAM has the following access exits for compression:

exk10 This is activated after a record in the original file has been read.

exk20 This is activated before a record in the FLAMFILE is written.

FLAM has the following user exits for decompression:

exd10 This is activated before a record in the decompressed file is written.

exd20 This is activated after a record in the FLAMFILE has been read.

The user exits must be linked to the subprogram interface (flam_upu.o) or to the record interface (flam_recu.o) and to the application program.This means that at present only one function can be used per user exit. Any user exits which are not required by an application program must be installed as dummy functions (please refer to sections 3.3 and 4.5 for details of how to link the programs).

Chapter 7 contains examples of user exits, as well as a program for compressing and decompressing files (similar to FLAM).

6.1.1 Programming the Access Exits

When programming user exits for file accesses, their execution environment must be taken into consideration. For example, operator interventions must be avoided, if their use in batch processing cannot be ruled out. Likewise, output to displays might result in undesirable distortions to FLAM's processing logs.

Storage areas made available through pointers must not be modified beyond the length specified in the call. Storage for inserted records must be requested and released by the exit itself.

The following pages describe the syntax of the interfaces between FLAM an the user exits in detail.

exd10

This user exit can be used during the decompression procedure for accesses to the decompressed file, in order to postprocess the records which must be output. It is activated by the flamup subprogram by means of the parameter exd10=exit-specification in the argument parameter_string.

It takes over control after the decompressed file is opened, before each write access and before the file is closed; the record which must be written is made available to it prior to write accesses. It sends a return code when it returns control to FLAM.

Syntax

```
void exd10 (long *functioncode,
            long *returncode,
            char **record_pointer,
            long *record_length,
            char *work);
```

Arguments

functioncode This argument transfers a function code to the user exit. The valid values are as follows:

- 0 First call for the file after open
- 4 Record made available in record buffer
- 8 Last call for the file before close

returncode The user exit sends a return code to FLAM with this argument (see separate description of return codes and their meanings).

record_pointer This argument contains a pointer to a character string. If a call is specified with function code 4, it contains the address of the record which must be written. This address can be modified by the user exit, for example in order to insert a record.

record_length If a call is specified with function code 4, this argument contains the length of the record which must be written. This length can be modified by the user exit, for example in order to insert a record.

work This argument is a 1 Kbyte area, which is made available by FLAM as a work area each time the user exit is invoked and which is not altered by FLAM between the calls. When it is invoked for the first time (function code 0), the name of the file is at the start of the area and the remainder of the area is initialized with binary zeros.

Return codes

Value	Meaning
0	Transfer record or no errors
4	Don't transfer record
8	Insert record. The buffer address and the length of the record which is to be inserted must be returned in the record_pointer and record_length arguments. The user exit is invoked again for the original record
12	Terminate compression procedure
16	Error in user exit; abnormal termination

exd20

This user exit can be used during the decompression procedure for accesses to the FLAMFILE, in order to preprocess the records which have been read.It can be activated either by the flamup subprogram or by the flmopf function of the record interface.

It is specified in the subprogram call by means of the parameter exd20=exit-specification in the argument parameter_string and in the flmopf function by means of the argument exitroutine_decomp.

It takes over control after the FLAMFILE is opened, after each read access and before the file is closed; the record which has been read is made available to it after read accesses. It sends a return code when it returns control to FLAM.

Syntax

```
void exd20 ( long *functioncode,
             long *returncode,
             char **record_pointer,
             long *record_length,
             char *work);
```

Arguments

functioncode This argument transfers a function code to the user exit. The valid values are as follows:

- 0 First call for the file after open
- 4 Record made available in record buffer
- 8 Last call for the file before close

returncode The user exit sends a return code to FLAM with this argument (see separate description of return codes and their meanings).

record_pointer This argument contains a pointer to a character string. The pointer is 4 bytes long and thus corresponds to a long integer. If a call is specified with function code 4, it contains the address of the record which has been read. This address can be modified by the user exit, for example in order to insert a record.

record_length If a call is specified with function code 4, this argument contains the length of the record which has been read. This length can be modified by the user exit, for example in order to insert a record.

work This argument is a 1 Kbyte area, which is made available by FLAM as a work area each time the user exit is invoked and which is not altered by FLAM between the calls. When it is invoked for the first time (function code 0), the name of the file is at the start of the area and the remainder of the area is initialized with binary zeros.

Return codes	Value	Meaning
	0	Transfer record or no errors
	4	Don't transfer record
	8	Insert record. The buffer address and the length of the record which is to be inserted must be returned in the record_pointer and record_length arguments. The user exit is invoked again for the original record
	12	Terminate compression procedure
	16	Error in user exit; abnormal termination

exk10

This user exit can be used during the compression procedure to process the records in the original file after they have been read. It is activated by the flamup subprogram by means of the parameter exk10=exit-specification in the argument parameter_string.

It takes over control after the original file is opened, after each read access and before the file is closed; the record which has been read is made available to it after read accesses. It sends a return code when it returns control to FLAM.

Syntax

```
void exk10 ( long *functioncode,
             long *returncode,
             char **record_pointer,
             long *record_length,
             char *work);
```

Arguments

functioncode This argument transfers a function code to the user exit. The valid values are as follows:

- 0 First call for the file after open
- 4 Record made available in record buffer
- 8 Last call for the file before close

returncode The user exit sends a return code to FLAM with this argument (see separate description of return codes and their meanings).

record_pointer This argument contains a pointer to a character string. The pointer is 4 bytes long and thus corresponds to a long integer. If a call is specified with function code 4, it contains the address of the record which has been read. This address can be modified by the user exit, for example in order to insert a record.

record_length If a call is specified with function code 4, this argument contains the length of the record which has been read. This length can be modified by the user exit, for example in order to insert a record.

work This argument is a 1 Kbyte area, which is made available by FLAM as a work area each time the user exit is invoked and which is not altered by FLAM between the calls. When it is invoked for the first time (function code 0), the name of the file is at the start of the area and the remainder of the area is initialized with binary zeros.

Return codes	Value	Meaning
	0	Transfer record or no errors
	4	Don't transfer record
	8	Insert record. The buffer address and the length of the record which is to be inserted must be returned in the record_pointer and record_length arguments. The user exit is invoked again for the original record
	12	Terminate compression procedure
	16	Error in user exit; abnormal termination

exk20

This user exit can be used during the compression procedure to process the records in the FLAMFILE before they are written. It can be activated either by the flamup subprogram or by the record interface.

It is specified in the subprogram call by means of the parameter exk20=exit specification in the argument parameter_string and in the flmopf function by means of the argument exitroutine_comp.

It takes over control after the FLAMFILE is opened, before each write access and before the file is closed; the record which must be written is made available to it before write accesses. It sends a return code when it returns control to FLAM.

Syntax

```
void exk20 ( long *functioncode,
             long *returncode,
             char **record_pointer,
             long *record_length,
             char *work);
```

Arguments

functioncode

This argument transfers a function code to the user exit. The valid values are as follows:

0	First call for the file after open
4	Record made available in record buffer
8	Last call for the file before close

returncode

The user exit sends a return code to FLAM with this argument (see separate description of return codes and their meanings).

record_pointer

This argument contains a pointer to a character string. The pointer is 4 bytes long and thus corresponds to a long integer. If a call is specified with function code 4, it contains the address of the record in the FLAMFILE which must be written.

record_length

If a call is specified with function code 4, this argument contains the length of the record which must be written.

work

This argument is a 1 Kbyte area, which is made available by FLAM as a work area each time the user exit is invoked and which is not altered by FLAM between the calls. When it is invoked for the first time (function code 0), the name of the file is at the start of the area and the remainder of the area is initialized with binary zeros.

Return codes	Value	Meaning
	0	Transfer record or no errors
	4	Don't transfer record
	8	Insert record. The buffer address and the length of the record which is to be inserted must be returned in the record_pointer and record_length arguments. The user exit is invoked again for the original record. During the decompression procedure, records which have been inserted in this way must be removed again by means of a complementary user exit
	12	Terminate compression procedure
	16	Error in user exit; abnormal termination

6.1.2 Linking File Access Exits to Application Programs

FLAM supports calling user exits for file accesses both on the subprogram interface level (flamup) as well as on the record interface level (flamrec), the latter supporting accesses to FLAMFILEs only.

In order to be able to process data records with FLAM by means of these user exits, the associated linked object files must contain all the access functions described above; any functions which are not required may be included as dummy functions.

User exits for file accesses are linked statically with FLAM as described in section 3.3, "Linking flamup ", as well as in section 4.5, "Linking Record Interface Functions to Application Programs".

6.2 The User Exit for Automatic Key Management (Key Exit)

The user exits for automatic key management can be activated via the subprogram interface flamup and - in contrast to the access exits - throu the flam command, but not through the record interface flamrec.

Another major difference is that a key exit routine is linked dynamically. It is created as a shared object, and both the name of the shared library and that of the called function can be chosen freely and are passed to FLAM as a parameter.

6.2.1 Programming the Key Exit

Key exit routines must also avoid interactive operations if they are meant for use in batch processing. Display messages can be synchronized with FLAM logging by use of the *message* and *msglen* arguments with function code -1 (version identification). At each execution, FLAM invokes the key exit routine once with this function code and logs the returned message text - if present - at an appropriate place. With other function codes, message texts should only be returned to indicate error causes.

Below is the detailed description of the interface between FLAM abd the key exit routine.

kmfunc

This user exit can be used at encryption or decryption to provide a password automatically. It is activated by the parameter -kmexit=*exit specification* in the flam command or in the parameter_string argument of the subprogram call flamup. The actual name used instead of *kmfunc* may be any valid function name.

It sends a return code when it returns control to FLAM.

Syntax

```
void kmfunc (signed long *functioncode,
             signed long *returncode,
             const unsigned long *parmlen,
             const unsigned char *param,
             unsigned long *datalen,
             unsigned char *data,
             unsigned long *ckylen,
             unsigned char *cryptokey,
             unsigned long *msglen,
             unsigned char *message);
```

Argumente

functioncode This argument transfers a function code to the user exit. The valid values are as follows:

-1	call for version identification
0	call for decryption
1	call for encryption

returncode The user exit sends a return code to FLAM with this argument (see separate description of return codes and their meanings).

parmlen When *functioncode* is 0 or 1, this argument contains the byte length of the data in the *param* argument. That may be an integer value between 0 and 256.

param When *parmlen* > 0, this argument contains the *exparm*-part of the *exit specification* in the -kmexit parameter (see description of the parameter -kmexit in section 2.3).

datalen When *functioncode* is 0 or 1, this argument contains the byte length of the data in the *data* argument. That may be an integer value between 0 and 512. For calls for encryption, it is set by the exit routine. For calls for decryption, it is set by FLAM.

data — When *datalen* > 0 with calls for encryption (*functioncode* 1), the exit routine returns here the data required for retrieving the password needed for decryption. FLAM stores these data in a user-specific FLAM file header. With calls for decryption (*functioncode* 0), FLAM returns here these data to the exit routine.

ckylen — After a successful call to the exit routine with *functioncode* 0 or 1, this argument contains the byte length of the password returned to FLAM in the *cryptokey* argument. Maximum key length is 64 bytes.

cryptokey — In this argument, FLAM receives after exit calls with *functioncode* 0 or 1 the password for encryption or decryption with the length specified in *ckylen*.

msglen — Upon return to FLAM, this argument indicates the byte length of the string in the *message* argument. Maximum is 128 bytes.

message — The exit routine may put here a message text which will be output to the log file by FLAM.

Return codes

Value	Meaning
0	Success
otherwise	Error

6.2.2 Creating a Key Exit

The key exit function may be given any valid function name. The function must reside in a shared library which will be opened by FLAM at execution time. For information regarding the generation of shared libraries please refer to the documentation of the compiler used.

FLAM (UNIX)

User Manual

Chapter 7:

Application Examples

7. Application Examples

This chapter illustrates FLAM's applications with numerous examples. These demonstrate the various options which are available for the flam command; C programs and excerpts from C programs show how the program interfaces can be used in practice. These examples are supplemented where necessary by instructions for linking the programs.

7.1 Commands

Only the parameters which are actually required are used in the examples. If the values of the parameters are required to vary from the default values, they must be specified explicitly.

7.2 Compressing

7.2.1 One File into One File

A file called test.dat is compressed and the compressed data is written in a file called test.cmp.

flam -compress -flamin=test.dat -flamfile=test. cmp

7.2.2 Several Files into One File

All the files which match the search pattern t*.dat are compressed and the compressed data is written in a file called test.cmp.

flam -compress -flamin=t*.dat -flamfile=test.cmp -attributes=all -show=all

-show=all causes all the compression information to be shown on the screen.

-attributes=all causes the names, file attributes and record attributes of the original files to be saved in the compressed file as well. They can then be shown on the screen with the following command

flam -decompress -show=attributes -flamfile=test.cmp

without creating the decompressed files.

7.2.3 Several Files into Several Separate Files

All the files which match the search pattern t*.dat are compressed. The compressed data of each file is written in a separate file, which is given the name of the original file and a cmp suffix:

flam -compress -flamin= t*.dat -flamfile=[dat=cmp]

If the default directory were also to contain a file called tvdaten.dat, for example, as well, the compressed data of this file would be saved in the FLAMFILE called tvcmpen.dat (see section 2.3.2, Output specifications).
The following notation should be preferred, to ensure that the desired substitution rule is actually applied to the suffix:

flam -compress -flamin=t*.dat -flamfile=[.dat=.cmp]

7.3 Decompressing

7.3.1 One File into One File

A file called test.cmp is decompressed and the decompressed data is written in a file called test.dat. The compressed file may comprise one or more original files.

flam -decompress -flamfile=test.cmp -flamout=test.dat

7.3.2 One Compressed File, Comprising Several Original Files, into Separate Files, Each Identical to One of the Original Files

A compressed file containing the data of several different original files, each with a FLAM file header (-attributes=all), is decompressed.
The decompressed data of each original file is written in a file with the same name and the same attributes:

flam -decompress -flamfile=test.cmp -flamout=[*]

7.3.3 Several Files into One File

All the files which match the search pattern t*.cmp are decompressed and the decompressed data is written in a file called test.dat:

flam -decompress -flamfile=t*.cmp -flamout=test.dat

7.3.4 Several Files into Several Separate Files

All the files which match the search pattern t*.cmp are decompressed. The decompressed data of each file is written in a separate file, which is given the same name and a dat suffix:

flam -decompress -flamfile=t*.cmp -flamout=[.cmp=.dat]

It should be noted in this connection that all the compressed data in a single FLAMFILE is also decompressed into a single file.
If -attributes=all was specified for the compression procedure, the following decompress command

flam -decompress -flamfile=t*.cmp -flamout=[*]

can be used to create decompressed files with the original names and the original attributes.

7.4 How to Use a Parameter File

The FLAM parameters do not necessarily need to be specified directly in the command. They can also be entered in a parameter file, which is then specified in the command instead:

flam -parfile=param.dat

The file called param.dat contains the following parameters:

For compression:
flamin=test.dat
flamfile=test.cmp
comp

For decompression:
flamfile=test.cmp
flamout=test.dat
decomp

7.5 How to Use the Subprogram Interface

Files can be compressed in a user program and decompressed again by means of the flamup subprogram interface. The call in a C user program is as follows:

flamup(id, rc, par_string, parlen);

id is the address of a 4 byte long field.

rc is the address of a 4 byte long numeric field, which contains the return code after flamup has been executed.

par_string is the address of a string which contains the parameters.

parlen is the address of a 4 byte long numeric field, which contains the length of par_string.

par_string contains, for example:

"**flamin=test.dat,flamfile=test.cmp,comp**"

In this case, parlen contains the value 38.

The sample program below uses the subprogram interface. The parameters are specified in the call in the same way as in the flam command.

This program can be used as a user program with user input/output routines and/or user exits:

```
/*

flamup call with user exit and/or user i/o

*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include "flamincl.h"

void main (int argc, char** argv)
{ char *id;                              /* ID */

  unsigned long retco;                   /* Return code */
  long ip, is;                           /* Integer */

  char *po, *pq;                         /* Pointer */
  unsigned char *pp;                     /* Pointer */

  char string[100];                      /* Character string */
  char parstring[1500];                  /* Parameter string */

  retco = FLAM_NORMAL;                   /* Return code = ok */
  pp = parstring;

  is = 0;
  ip = 0;
  while (++ip < argc)
  { po = argv[ip];                       /* Parameter after input area */
    pq = &string[0];
    while (*po != 0x00)
    { if ((*po != '-') && (*po != '+'))
        *pq++ = *po++;
      else
        po++;
    };
    *pq = 0x00;                          /* Contains parameter lists ? */
    po = &string[0];
    while ((*po != 0x00) && (*po != ',') && (*po != ' '))
      po++;
```

```
  if (*po == 0x00)                    /* Parameter without lists */
  { po = &string[0];                  /* After parameter string */
    while (*po != 0x00)
    { *pp++ = *po++;
        is++;
    }
  }
  else
  { po = &string[0];                  /* Parameter with list */
    do                                /* "d1 d2 ..." */
    { *pp++ = *po;                    /* "d1,d2,..."     */
      is++;                           /* d1,d2,... */
    } while (*po++ != '=');           /* After parameter string */
    *pp++ = '(';
    is++;
    if (*po == '"')
      po++;
    while (*po != 0x00)
    { if ((*po != ',') && (*po != '"') && (*po != ' ') )
      { *pp++ = *po++;
        is++;
      }
      else
      { if (*po == '"')
          po++;
        else
        { *pp++ = ',';
          po++;
          is++;
        }
      }
    }

    if (*(pp - 1) == ',')
    { pp--;
      is--;
    }
    *pp++ = ')';
    is++;
  }
  *pp++ = ',';
  is++;
}
if (is  0)
{ *--pp = ' ';
  is--;
}
```

```
/*

         flamup call

*/

  flamup(&id, &retco, parstring, &is);

  put_flmsg(retco);                        /* Return code output */

  return retco;

}
```

7.6 How to Use the Record Interface

7.6.1 Compressing a File

Data records are read in a user program and transferred to FLAM's record interface for compression. The information about the original file and the information which is generated by the user is saved in the file header.

The names of the input and output files are entered interactively.

```
/*

Example: Compress a file
The file header is written with a common part and a user part

*/

#include <stdio.h>
#include <time.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <fcntl.h>
extern int errno;
extern int sys_nerr;

#include "flamincl.h"

int main()
{ char *flmid;                          /* Record interface ID */
  char kname[100];                       /* Name of compressed file */
  char oname[100];                       /* Name of original file */
  long flcode;                          /* FLAM code */
  long modus;                            /* Compression mode */
  long statis;                           /* Statistics */
  long opmode;                           /* Open mode */
  long blkmode;                          /* Block mode */
  long header;                           /* File header */
  long prctrl;                           /* Feed control character */
  long reclength;                        /* Record length */
  long namlen;                           /* Length of file name */
  long sattrlen;                    /* Length of system-specific information */
  union { char c[4];
          char *p;
        } sysattr;                       /* System-specific information */
  char datrec[2048];                     /* Data area */
  char system[2];                        /* Operating system */
```

```
long sanz;                              /* Max. no. of records / block */
unsigned long maxb;                     /* Max. buffer size in KB */
long lstpar;                            /* Last parameter */
                                        /* Compressed file */
long blksk;                             /* Block size */
long recfk;                             /* Record format */
long recsk;                             /* Record size */
long devk;                              /* Device */
long orgak;                             /* Organization */
union { char c[4];
        unsigned long i;
      } recdelk;                        /* Record delimiter */
long cldispk;                           /* Close disposition */
struct kd keydesck;                     /* Key description */
                                        /* Original file */
long devo;                              /* Device */
long orgao;                             /* Organization */
long blkso;                             /* Block size */
long recfo;                             /* Record format */
long recso;                             /* Record size if fixed */
union { char c[4];
        unsigned long i;
      } recdelo;                        /* Record delimiter */
long cldispo;                           /* Close disposition */
struct kd keydesco;                     /* Key description */

char exk20[34];                         /* Name of exit routine */
char exd20[34];                         /* Name of exit routine */

char *ptr;                              /* Pointer */
long il;                                /* Integer */
unsigned long rc;                       /* Return code */
unsigned long cputime;                  /* CPU time */

unsigned long zks;                      /* Record counter */
unsigned long zkb;                      /* Byte counter */
unsigned long zkbofl;                  /* Byte counter overflow */
unsigned long zunks;                    /* Record counter */
unsigned long zunkb;                    /* Byte counter */
unsigned long zunkbofl;                /* Byte counter overflow */

FILE *infile;                          /* Input file */

long uattrlen;                          /* Length of user information */
static char usrattr[50] = {"The file contains data belonging to
application xyz"};

                                        /* Enter file name */
printf("File name input: ");
scanf("%s", oname);
printf("File name output: ");
scanf("%s", kname);
                                        /* Open original file */
infile = fopen(oname, "r");
if (infile == NULL)
  exit(1);
```

```
                                        /* Open FLAM */
opmode = FLAM_C_OPEN_OUTPUT;            /* Compress */
statis = FLAM_C_STATISTIC;              /* Statistics */
lstpar = FLAM_C_MORE_PARAMETER;
flmopn(&flmid, &rc, &lstpar, &opmode, kname, &statis);
if (rc != FLAM_NORMAL)
  closinput(rc, infile);
                                    /* Specifications for compressed file */
orgak = FLAM_C_ORG_SEQ;
recfk = FLAM_C_RECFRM_FIX;
recsk = 512;
blksk = 0;
cldispk = FLAM_C_CLOSE_REWIND;
devk = FLAM_C_DEVICE_DISK;
keydesck.keyparts = 0;
il = 100;
flmopd(&flmid, &rc, &lstpar, &il, kname, &orgak, &recfk,  &recsk,
       recdelk.c, &keydesck, &blksk, &cldispk, &devk);
if (rc != FLAM_NORMAL)
  closinput(rc, infile);
                                        /* Compression specifications */
il = FLAM_C_VERSION;                    /* Version 2 */
flcode = FLAM_C_CHAR_ASCII;            /* Code of compressed file */
modus = FLAM_C_MODUS_CX8;               /* EIGHT_BIT mode */
maxb = 32768;                           /* Buffer size */
sanz = 255;                             /* No. of records per block */
header = FLAM_C_FILEHEADER;             /* File header */
blkmode = FLAM_C_BLOCKMODE;             /* FLAMFILE blocking mode */
keydesco.keyparts = 0;
exk20[0] = ' ';                         /* No exit routine */
exd20[0] = ' ';
flmopf(&flmid, &rc, &il, &flcode, &modus, &maxb, &header,
       &sanz, &keydesco, &blkmode, exk20, exd20);
if (rc != FLAM_NORMAL)
  closinput(rc, infile);
                                        /* Output file header */
namlen = strlen(oname);
orgao = FLAM_C_ORG_SEQ;                 /* Organization seq. */
recfo = FLAM_C_RECFRM_STREAM;           /* Record format stream, text file */
recso = 0;
blkso = 0;
prctrl = FLAM_C_PRINT_CTRL_NONE;
system[0] = FLAM_C_SYSTEM_COMP;         /* Computer / processor */
system[1] = FLAM_C_SYSTEM_OS;           /* Operating system */
flmphd(&flmid, &rc, &namlen, oname, &orgao, &recfo, &recso,
       recdelo.c, &keydesco, &blkso, &prctrl, system, &lstpar);
if (rc != FLAM_NORMAL)
  closfiles(rc, infile, &flmid);
                                      /* Output user-specific file header */
uattrlen = strlen(usrattr);
flmpuh(&flmid, &rc, &uattrlen, usrattr);
if (rc != FLAM_NORMAL)
  closfiles(rc, infile, &flmid);
```

```
                                        /* Read and compress file */
  il = FLAM_NORMAL;
  ptr = fgets(datrec, 2048, infile);
  if (ptr == NULL)
    if (errno == 0)
      il = FLAM_EOF;
    else
    { rc = errno + FLAM_IO;
        closfiles(rc, infile, &flmid);
    }
  while (il == FLAM_NORMAL)
  { reclength = strlen(datrec) - 1;
                                        /* Output compressed file */
    flmput(&flmid, &rc, &reclength, datrec);
    if (rc != FLAM_NORMAL)
      closfiles(rc, infile, &flmid);
                                        /* Read original file */
    ptr = fgets(datrec, 2048, infile);
    if (ptr == NULL)
      if (errno == 0)
        il = FLAM_EOF;
      else
      { rc = errno + FLAM_IO;
        closfiles(rc, infile, &flmid);
      }
  }

                                        /* Exit FLAM */
  flmcls(&flmid, &rc, &cputime, &zunks, &zunkb, &zunkbofl,
         &zks, &zkb, &zkbofl);
                                        /* Output statistics */
  printf("\nNo. of non-compressed records: %d\n", zunks);
  printf("No. of non-compressed bytes:  %d\n", zunkb);
  printf("\nNo. of non-compressed records: %d\n", zks);
  printf("No. of non-compressed bytes:  %d\n", zkb);

  if (rc != FLAM_NORMAL)
    closinput(rc, infile);

                                        /* Close original file */
  rc = fclose(infile);

  return rc;
}

/*
       Close files, exit program
*/
```

```
void closfiles(unsigned long rc, FILE *infile, char **flmid)
{ long il;
  unsigned long cputime;                  /* CPU time */
  unsigned long zks;                      /* Record counter */
  unsigned long zkb;                      /* Byte counter */
  unsigned long zkbofl;                  /* Byte counter overflow */
  unsigned long zunks;                    /* Record counter */
  unsigned long zunkb;                    /* Byte counter */
  unsigned long zunkbofl;                /* Byte counter overflow */

                                          /* Exit FLAM */
  flmcls(flmid, &il, &cputime, &zunks, &zunkb, &zunkbofl,
         &zks, &zkb, &zkbofl);
                                          /* Output statistics */
  if (il == FLAM_NORMAL)
  { printf("\nNo. of non-compressed records: %d\n", zunks);
    printf("No. of non-compressed bytes:  %d\n", zunkb);
    printf("\nNo. of non-compressed records: %d\n", zks);
    printf("No. of non-compressed bytes:  %d\n", zkb);
  }
  closinput(rc, infile);
}

/*
Close original file, exit program
*/

void closinput(unsigned long rc, FILE *infile)
{ long il;
                                  /* Close original file */
  il = fclose(infile);
  put_flmsg(rc);

  exit(rc);
}
```

7.6.2 Decompressing a File

The records of a file which have been decompressed by FLAM are recalled. The file header, which comprises a common part and a user-specific part, is read. The names of the input and output files are entered interactively.

```
/*

Example: Decompress a file
The file header is read with a common part and a user part

*/

#include <stdio.h>
#include <time.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <fcntl.h>
extern int errno;
extern int sys_nerr;

#include "flamincl.h"

int main()
{ char *flmid;                    /* Record interface ID */
  char kname[100];                 /* Name of compressed file */
  char oname[100];                 /* Name of original file */
  char dname[100];                 /* Name of decompressed file */

  long flcode;                    /* FLAM code */
  long modus;                      /* Compression mode */
  long statis;                     /* Statistics */
  long opmode;                     /* Open mode */
  long blkmode;                    /* Block mode */
  long header;                     /* File header */
  long prctrl;                     /* Feed control character */
  long version;                    /* FLAM version */
  long reclength;                  /* Record length */
  long buflength;                 /* Buffer length */
  long namlen;                     /* Length of file name */
  long sattrlen;                   /* Length of system-specific information */

  char sysattr[2048];              /* System-specific information */
  char datrec[2048];               /* Data area */
  char system[2];                  /* Operating system */

  long sanz;                       /* Max. no. of records / block */
  unsigned long maxb;              /* Max. buffer size in KB */
  long lstpar;                     /* Last parameter */
```

```
                                /* Compressed file */
long blksk;                     /* Block size */
long recfk;                     /* Record format */
long recsk;                     /* Record size */
long devk;                      /* Device */
long orgak;                     /* Organization */
union { char c[4];
        unsigned long i;
      } recdelk;                /* Record delimiter */
long cldispk;                   /* Close disposition */
struct kd keydesck;             /* Key description */
                                /* Decompressed file */
long devd;                      /* Device */
long orgad;                     /* Organization */
long blksd;                     /* Block size */
long recfd;                     /* Record format */
long recsd;                     /* Record size if fixed */
union { char c[4];
        unsigned long i;
      } recdeld;                /* Record delimiter */
long cldispd;                   /* Close disposition */
struct kd keydescd;             /* Key description */
                                /* Original file */
long blkso;                     /* Block size */
long recfo;                     /* Record format */
long recso;                     /* Record size */
long devo;                      /* Device */
long orgao;                     /* Organization */
union { char c[4];
        unsigned long i;
      } recdelo;                /* Record delimiter */
long cldispo;                   /* Close disposition */
struct kd keydesco;             /* Key description */

char exk20[34];                 /* Name of exit routine */
char exd20[34];                 /* Name of exit routine */

long il, ir;                    /* Integer */
unsigned long rc;               /* Return code */
unsigned long cputime;          /* CPU time */

unsigned long zks;              /* Record counter */
unsigned long zkb;              /* Byte counter */
unsigned long zkbofl;           /* Byte counter overflow */
unsigned long zunks;            /* Record counter */
unsigned long zunkb;            /* Byte counter */
unsigned long zunkbofl;         /* Byte counter overflow */

FILE *outfile;                  /* Output file */

long uattrlen;
char usrattr[1000];
```

```
                                /* Enter file name */
printf("File name input: ");
scanf("%s", kname);

printf("File name output: ");
scanf("%s", dname);

                                /* Initialize FLAM */
opmode = FLAM_C_OPEN_INPUT;     /* Decompress */
statis = FLAM_C_STATISTIC;      /* Statistics */
lstpar = FLAM_C_MORE_PARAMETER;
flmopn(&flmid, &rc, &lstpar, &opmode, kname, &statis);
if (rc != FLAM_NORMAL)
{ put_flmsg(rc);
  exit (rc);
}
                                /* Specifications for compressed file */
il = 100;
cldispk = FLAM_C_CLOSE_REWIND;  /* Close disposition */
flmopd(&flmid, &rc, &lstpar, &il, kname, &orgak, &recfk, &recsk,
       recdelk.c, &keydesck, &blksk, &cldispk, &devk);
if (rc != FLAM_NORMAL)
{ put_flmsg(rc);
  exit (rc);
}
                                /* Compression specifications */
header = FLAM_C_FILEHEADER;     /* Read file header */
keydesco.keyparts = 0;
exk20[0] = ' ';                 /* No exit routine */
exd20[0] = ' ';
flmopf(&flmid, &rc, &version, &flcode, &modus, &maxb, &header, &sanz,
       &keydescd, &blkmode, exk20, exd20);
if (rc != FLAM_NORMAL)
{ put_flmsg(rc);
  exit (rc);
}
                                /* Read file header */
                                /* Common information */
namlen = 100;
flmghd(&flmid, &rc, &namlen, oname, &orgao, &recfo, &recso,
       recdelo.c, &keydesco, &blkso, &prctrl, system);
if ((rc != FLAM_NORMAL) && (rc != FLAM_NO_FILEHEADER))
  closflam(rc, &flmid);
                                /* VAX/VMS system ? */
if (rc != FLAM_NO_FILEHEADER)
{
                                /* User-specific information */
  uattrlen = 1000;
  flmguh(&flmid, &rc, &uattrlen, usrattr);
  if ((rc != FLAM_NORMAL) && (rc != FLAM_NO_FILEHEADER))
    closflam(rc, &flmid);
}
                                /* Open decompressed file */
outfile = fopen(dname, "w");
if (outfile == NULL)
  exit(1);
```

```
                                    /* Decompress and write file */
  buflength = 2048;
  flmget(&flmid, &rc, &reclength, datrec, &buflength);
  if (rc != FLAM_NORMAL)
  { if (rc == FLAM_EOF)
      rc = FLAM_NORMAL;
    closfiles(rc, outfile, &flmid);
  }
  il = FLAM_NORMAL;
  while (il == FLAM_NORMAL)
  { datrec[reclength++] = 0x0a;
    datrec[reclength] = 0x00;
    ir = fputs(datrec, outfile);
    if (ir != reclength)
    { rc = FLAM_WRITE_ERR;
      closfiles(rc, outfile, &flmid);
    }
                                    /* Read decompressed record */
    flmget(&flmid, &rc, &reclength, datrec, &buflength);
    if (rc != FLAM_NORMAL)
    { if (rc == FLAM_EOF)
        il = rc;
      else
        closfiles(rc, outfile, &flmid);
    }
  }

                                    /* Close decompressed file */
  rc = close(outfile);

                                    /* Exit FLAM */
  flmcls(&flmid, &rc, &cputime, &zunks, &zunkb, &zunkbofl, &zks,
         &zkb, &zkbofl);
                                    /* Output statistics */
  printf("\nNo. of non-compressed records: %d\n", zunks);
  printf("No. of non-compressed bytes:  %d\n", zunkb);
  printf("\nNo. of compressed records: %d\n", zks);
  printf("No. of compressed bytes:  %d\n", zkb);
  put_flmsg(rc);
  return rc;
}

/*
Close decompressed file, exit FLAM
*/

closfiles(unsigned long rc, FILE *outfile, char **flmid)
{ long il;
                                    /* Close decompressed file */
  il = close(outfile);

  closflam(rc, flmid);
}
```



```
/*
Exit FLAM, exit program
*/

closflam(unsigned long rc, char **flmid)
{ long il;
  unsigned long cputime;           /* CPU time */
  unsigned long zks;               /* Record counter */
  unsigned long zkb;               /* Byte counter */
  unsigned long zkbofl;           /* Byte counter overflow */
  unsigned long zunks;             /* Record counter */
  unsigned long zunkb;             /* Byte counter */
  unsigned long zunkbofl;         /* Byte counter overflow */

                                   /* Exit FLAM */
  flmcls(flmid, &il, &cputime, &zunks, &zunkb, &zunkbofl,
         &zks, &zkb, &zkbofl);
                                   /* Output statistics */
  if (il == FLAM_NORMAL)
  { printf("\nNo. of non-compressed records: %d\n", zunks);
    printf("No. of non-compressed bytes:  %d\n", zunkb);
    printf("\nNo. of non-compressed records: %d\n", zks);
    printf("No. of non-compressed bytes:  %d\n", zkb);
  }

  if (rc == FLAM_NORMAL)
    rc = il;
                                   /* Convert return */
  put_flmsg(rc);
  exit (rc);
}
```

7.7 User-Defined Input/Output

The examples below describe the interfaces for the user-defined input and output functions.

The calls, the necessary parameters and the possible values are specified for each function. Input and output errors are not shown.

7.7.1 Opening a File

```
/*

Open a file to read or write (user_io)

*/

#include "flamincl.h"

void usropn(workio, retco, openmode, linkname, fcbtype, recform, recsize,
            blksize, keydesc, device, recdelim, padchar, prcrtl,
            closdisp, access,  namelen, filename)

FLAM_PPCHAR workio;     /*    File ID */
FLAM_PLONG *retco;      /*    Return code */
FLAM_PLONG *openmode;   /*    Processing mode */
                        /*    0 = input (seq. read)
                              1 = output (seq. write)
                              2 = inout (read and write with key)
                                  (existing file)
                              3 = outin (write and read with
                                  key)
                                  (new file) */
FLAM_PPCHAR linkname;    /*   Link name / file name */
FLAM_PLONG *fcbtype;     /*   Organization */
                         /*   0,  8, 16, ... = sequential
                              1,  9, 17, ... = index-sequential
                              2, 10, 18, ... = relative
                              3, 11, 19, ... = direct access
                              5, 13, 21, ... = library
                              6, 14, 22, ... = physical
                                     negative = use system-specific
                                                attributes
                             (if open = output, sattrlen not equal to 0)  */
```

```
FLAM_PLONG *recform;     /*   Record format */
                         /*   0, 8, 16, ... = variable
                              8  = blocked,
                              16 = blocked/spanned
                              24 = VFC
                              32 = 2 byte size field
                              40 = 4 byte ASCII size field
                              48 = 4 byte EBCDIC size field
                              1, 9, 17, ... = fixed
                              8  = blocked,
                              16 = blocked/standard
                              2, 10, 18, ... = undefined
                              18 = EAF
                              3, 11, 19, ... = stream
                              negative = use system-specific
                                         attributes
                                         (if open = output, sattrlen
                                         not equal to 0)  */
FLAM_PLONG *recsize;     /*   Record size */
                         /*   0 - 32767
                              recform v: Max. record size / 0
                              recform f: Record size
                              recform u: Max. record size / 0
                              recform s: Max. record size / 0 */
FLAM_PLONG *blksize;     /*   Block size */
                         /*   0 = unblocked
                              1 - 32767
                              negative = use system-specific
                                         attributes
                                         (if open = output, sattrlen
                                         not equal to 0)  */
PSTRKD *keydesc;         /*   Key description */
FLAM_PLONG *device;      /*   Device type */
                         /*   0 = hard disk / unknown
                              1 = tape
                              2 = diskette
                              3 = streamer
                              7/15/23 = user i/o
                              negative = use system-specific
                                         attributes
                                         (if open = output, sattrlen
                                         not equal to 0) */
FLAM_PCHAR recdelim;     /*   Record delimiter, terminated with '\0'
                              If vfc, 1st byte contains header
                              length */
FLAM_PCHAR padchar;      /*   Padding character */
FLAM_PLONG *prcrtl;      /*   Feed control character */
FLAM_PLONG *closdisp;    /*   Close disposition */
                         /*   0 = rewind
                              1 = unload
                              2 = retain / leave */
FLAM_PLONG *access;      /*   Access method */
                         /*   0 = logical
                              1 = physical (block-oriented)
                              2 = mixed (block access with record transfer) */
```

```
FLAM_PLONG *namelen;   /*    Length of file name (expanded name) */
FLAM_PPCHAR filename;  /*    File name (expanded name) */

{

*retco = 0;

/*
Open file
*/
.
.
.
ende: ;

return;

}
```

7.7.2 Reading a File

```
/*

  Read file (user_io)

*/

#include "flamincl.h"

void usrget(workio, retco, reclen, record, buflen)

FLAM_PPCHAR workio;     /*   File ID */
FLAM_PLONG *retco;      /*   Return code */
FLAM_PLONG *reclen;     /*   Record length in bytes */
FLAM_PPCHAR record;     /*   Record */
FLAM_PLONG *buflen;     /*   Length of record buffer in bytes */

{

*retco = 0;

/*
        Read record
*/
.
.
.
ende:

return;

}
```

7.7.3 Writing a File

```
/*

  Write file (user_io)

*/

#include "flamincl.h"

void usrput(workio, retco, reclen, record)

FLAM_PPCHAR workio;     /*   File ID */
FLAM_PLONG *retco;      /*   Return code */
FLAM_PLONG *reclen;     /*   Record length in bytes */
FLAM_PPCHAR record;     /*   Record */

{

*retco = 0;

/*
        Write record
*/
.
.
.
ende:

return;

}
```

7.7.4 Closing a File

```
/*

  Close file (user_io)

*/

#include "flamincl.h"

void usrcls(workio, retco)

FLAM_PPCHAR workio;     /*   File ID */
FLAM_PLONG *retco;      /*   Return code */

{

*retco = 0;

/*
        Close file
*/
.
.
.
ende:

return;

}
```

7.8 User Exits

7.8.1 Selecting Records of a Particular Type from a File with exk10 and Compressing Them

FLAM reads the records in a file and transfers them to exk10 for processing. The record type (1st character in the record) is verified by exk10. Records whose type is "1" are returned to FLAM for compression (returncode=0); all other records are not processed any further (returncode=4).

```
/*

       Select records of a particular type from a file
       with exitk10 and compress them

*/

exk10(functioncode, returncode, record_pointer, record_length, work)

long *functioncode;          /* Function code */
long *returncode;            /* Return code */
char **record_pointer;       /* Record pointer */
long *record_length;         /* Record length */
char *work;                  /* Work area */

{

*returncode = 0;

                             /* Call after open or before close */
if ((*functioncode == 0) ||  (*functioncode == 8))
  return;

if (**record_pointer != '1')
                             /* Records whose type is 1 (1st character
                                in record) are compressed; all
                                other records are skipped */
  *returncode = 4;

return;

}
```


7.8.2 Processing Records in a Compressed File with exd10

The decompressed records are not written in the decompressed file, but are transferred by FLAM to exd10 for processing. exd10 processes the records (screen output shown here) and does not return them for writing (returncode=4).

```
/*

       Process records in a compressed file with exitd10.
       The decompressed records are not written in the
       decompressed file.

*/

#include  <stdio.h>

exd10(functioncode, returncode, record_pointer, record_length, work)

long *functioncode;          /* Function code */
long *returncode;            /* Return code */
char **record_pointer;       /* Record pointer */
long *record_length;         /* Record length */
char *work;                  /* Work area */

{

*returncode = 0;

if ((*functioncode == 0) ||  (*functioncode == 8))
  return;


/*

File record processed (shown)

*/

*(*record_pointer + *record_length) = 0x00;
*work = printf("%s\n", *record_pointer);
*returncode = 4;             /* No errors, don't transfer record */

return;

}
```

7.8.3 Swapping Two Bytes with exk20 during Compression

Two bytes are swapped in each record, to provide additional protection for the compressed file. If the data were to be decompressed without swapping the bytes back again, an error would occur.

The swapped bytes are returned to their original positions using the same program during the decompression procedure.

```
/*

       Example of exitk20 (and exitd20)
       Swap two bytes during compression procedure.

       The swapped bytes are returned to their original
      positions using the same program
      during the decompression procedure.
       The program call then becomes exd20 instead of exk20.

*/

exk20(functioncode, returncode, record_pointer, record_length, work)

long *functioncode;          /* Function code */
long *returncode;            /* Return code */
char **record_pointer;       /* Record pointer */
long *record_length;         /* Record length */
char *work;                  /* Work area */

{

*returncode = 0;             /* No errors, transfer record */

if ((*functioncode == 0) ||  (*functioncode == 8))
  return;

if (*record_length > 16)     /* Swap bytes 16 and 17 */
  { *work = *(*record_pointer + 15);
    *(*record_pointer + 15) = *(*record_pointer + 16);
    *(*record_pointer + 16) = *work;
  };

return;

}
```

7.8.4 Swapping Two Bytes with exd20 during Decompression

The bytes which were swapped with exk20 during the compression procedure are returned to their original positions (see section 7.8.3).

The program is the same as for compression with exitk20.

```
/*

     Example of exitd20
     Swap two bytes during decompression procedure.

     Same program as for compression with exitk20.

*/

exd20(functioncode, returncode, record_pointer, record_length, work)

long *functioncode;          /* Function code */
long *returncode;            /* Return code */
char **record_pointer;       /* Record pointer */
long *record_length;         /* Record length */
char *work;                  /* Work area */

{

*returncode = 0;             /* No errors, transfer record */

if ((*functioncode == 0) || (*functioncode == 8))
  return;

if (*record_length > 16)     /* Swap bytes 16 and 17 */
  { *work = *(*record_pointer + 15);
    *(*record_pointer + 15) = *(*record_pointer + 16);
    *(*record_pointer + 16) = *work;
  };

return;

}
```

7.8.5 Automatic Key Management Function

When using encryption/decryption, the -kmexit parameter can be used to make FLAM activate a function that automatically generates or retrieves the required passwords. Such a function (as a C language program) might look like this:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/types.h>
#include <sys/stat.h>

#define  KMX_KEY_LENGTH        8
#define  KMX_ID_INFO           0xFFFFFFFF
#define  KMX_ENCRYPT           1
#define  KMX_DECRYPT           0
void samplex1                 /***********************************************/
(                            /* To generate a password (fuco = 1), this exit */
  signed   long *fuco,       /* routine randomly positions into a text file  */
  signed   long *retco,      /* and extracts a string of length 8. It passes */
  unsigned long *parmlen,    /* back the string in cryptokey and the blurred */
  unsigned char *param,      /* random position in data.                     */
  unsigned long *datalen,    /* To retrieve a password (fuco = 0), the read  */
  unsigned char *data,       /* position is derived from data and the string */
  unsigned long *ckylen,     /* found there is passed back in cryptokey.     */
  unsigned char *cryptokey,/* With fuco = -1, only a text line identifying */
  unsigned long *msglen,     /* the routine is passed back in message.       */
  unsigned char *message      /***********************************************/
)
{ const char          exit_id[] = "Key Management Exit Version 1.0.0";
  const char          ffkmx  [] = "/flam/ffkmx";
  const char          errmsg1[] = "Can't get info on file /flam/ffkmx";
  const char          errmsg2[] = "Can't open file /flam/ffkmx";
  const char          errmsg3[] = "Positioning in file /flam/ffkmx failed";
  const char          errmsg4[] = "Reading file /flam/ffkmx failed";
  const char          errmsg5[] = "Function code %d not supported";
  const char          errmsg6[] = "File /flam/ffkmx too small";
  unsigned long       count1   ;
  unsigned long       count2   ;
  unsigned long       rnd_nbr  ;
  unsigned long       read_pos ;
  FILE                *khandle ;
  time_t              unixtime ;
  struct tm           *p_tm    ;
  struct stat         filestat ;

  *msglen = 0;                                  /*                          */
  *retco  = stat(ffkmx, &filestat);            /* Determine file size     */
  if (*retco != 0)                              /*                          */
  { strcpy(message, (char*)errmsg1);            /*                          */
    *msglen = strlen(message);                  /*                          */
    return;                                     /*                          */
  }                                             /*                          */
                                                /*                          */
  if (filestat.st_size < 256)                   /* Check file's size       */
  { strcpy(message, (char*)errmsg6);            /*                          */
    *msglen = strlen(message);                  /*                          */
    return;                                     /*                          */
  }                                             /*                          */
  khandle = fopen(ffkmx, "rb");                 /* Open text file          */
```

```
if (khandle == NULL)                              /*                          */
{ strcpy(message, (char*)errmsg2);                /*                          */
  *msglen = strlen(message);                      /*                          */
  return;                                         /*                          */
}                                                 /*                          */
switch (*fuco)                                    /*                          */
{ case KMX_ENCRYPT  :                             /* For Encryption:          */
         time(&unixtime);                         /*                          */
         srandom(unixtime % 0x00010001L);         /*   generate random #      */
         read_pos = rnd_nbr                       /*                          */
                  = random() % filestat.st_size;/* as read position and  */
         rnd_nbr  ^= 0x55555555L;                 /*   obscure it             */
         data[3]   = (char)rnd_nbr;               /*   save modified #       */
         rnd_nbr >>= 8;                           /*                          */
         data[2]   = (char)rnd_nbr;               /*                          */
         rnd_nbr >>= 8;                           /*                          */
         data[1]   = (char)rnd_nbr;               /*                          */
         rnd_nbr >>= 8;                           /*                          */
         data[0]   = (char)rnd_nbr;               /*                          */
         *datalen  = 4;                           /*                          */
         break;                                   /*                          */
                                                  /*                          */
  case KMX_DECRYPT:                               /* For Decryption           */
         rnd_nbr   = (unsigned long)data[0];      /*   retrieve saved number */
         rnd_nbr <<= 8;                           /*                          */
         rnd_nbr  += (unsigned long)data[1];      /*                          */
         rnd_nbr <<= 8;                           /*                          */
         rnd_nbr  += (unsigned long)data[2];      /*                          */
         rnd_nbr <<= 8;                           /*                          */
         rnd_nbr  += (unsigned long)data[3];      /*                          */
         read_pos  = rnd_nbr ^ 0x55555555L;       /*   undo obscuring         */
                                                  /*                          */
         break;                                   /*                          */
                                                  /*                          */
  case KMX_ID_INFO:                               /* For Info request         */
         strcpy(message, (char*)exit_id);         /*   pass ID_string         */
         *msglen = strlen(message);               /*                          */
         *retco = 0;                              /*                          */
         return;                                  /*                          */
                                                  /*                          */
  default           :                             /* For all other codes      */
         sprintf(message, errmsg5, *fuco);        /*   issue error            */
         *msglen = strlen(message);               /*                          */
         *retco = -1;                             /*                          */
         return;                                  /*                          */
}                                                 /*                          */
*retco  = fseek(khandle, read_pos, SEEK_SET); /* Position on read pos.    */
if (*retco != 0)                                  /*                          */
{ strcpy(message, (char*)errmsg3);                /*                          */
  *msglen = strlen(message);                      /*                          */
  return;                                         /*                          */
}                                                 /*                          */
```

```
  count1 = filestat.st_size - read_pos;          /* Check if file wrap      */
  if (count1 >= KMX_KEY_LENGTH)                   /* necessary               */
  { *retco = fread(cryptokey, 1,                  /* if not, read full key   */
                   KMX_KEY_LENGTH, khandle);      /*                         */
    if (*retco != KMX_KEY_LENGTH)                 /*                         */
    { strcpy(message, (char*)errmsg4);            /*                         */
      *msglen = strlen(message);                  /*                         */
      *retco = -1;                                /*                         */
      return;                                     /*                         */
    }                                             /*                         */
    *retco = 0;                                   /*                         */
  }                                               /*                         */
  else                                            /*                         */
  { *retco = fread(cryptokey, 1, count1, khandle);/*if yes, read first part */
    if (*retco != count1)                         /*                         */
    { strcpy(message, (char*)errmsg4);            /*                         */
      *msglen = strlen(message);                  /*                         */
      *retco = -1;                                /*                         */
      return;                                     /*                         */
    }                                             /*                         */
    read_pos = 0;                                 /*                         */
    *retco  = fseek(khandle, read_pos, SEEK_SET);/* reposition on beginning */
    if (*retco != 0)                              /* (wrap around)           */
    { strcpy(message, (char*)errmsg3);            /*                         */
      *msglen = strlen(message);                  /*                         */
      return;                                     /*                         */
    }                                             /*                         */
    count2 = KMX_KEY_LENGTH - count1;             /*                         */
    *retco = fread(&cryptokey[count1], 1,         /* then read second part   */
                   count2, khandle);              /*                         */
    if (*retco != count2)                         /*                         */
    { strcpy(message, (char*)errmsg4);            /*                         */
      *msglen = strlen(message);                  /*                         */
      *retco = -1;                                /*                         */
      return;                                     /*                         */
    }                                             /*                         */
  }                                               /*                         */
  cryptokey[KMX_KEY_LENGTH] = '\0';               /* Terminate string        */
  *ckylen                   = KMX_KEY_LENGTH;     /* Set length              */
  fclose(khandle);                                /* Close file             */
  return;                                         /*                         */
}                                                 /*                         */
```

This function responds to requests passed via the user exit for automatic key management. For FLAM to be able to activate it, it must be a member of a shared library located in the lib-subdirectory of FLAM's installation directory. Assuming that this library is named libuserex.so, then, to have this function manage the passwords automatically, the flam command or the command string passed to flamup must contain

-kmexit=samplex1(libuserex)

For encryption (compression), a password is then generated automatically and passed to FLAM along with the information required to retrieve that password. FLAM stores this information in a user-specific file header. For decryption (decompression), FLAM passes this information back to this function and receives in return the matching password.

FLAM (UNIX)

User Manual

Appendix A:

Return Codes

A. Return Codes

Return Code	Symbolic Name - Meaning
-1	FLAM_FCT_NOT_EXIST **Meaning:** Function not available/not allowed/does not exist (changed by flamup to 999)
0	FLAM_NORMAL **Meaning:** No errors
1	FLAM_RECORD_SHORTENED **Meaning:** Decompressed record shortened (record interface warning)
2	FLAM_EOF **Meaning:** End of file
4	FLAM_RECORD_LENGTHENED **Meaning:** Decompressed record(s) lengthened
6	FLAM_NEW_FILE **Meaning:** Start of new file
7	FLAM_NO_PW **Meaning:** Password must be specified
9	FLAM_NO_FILEHEADER **Meaning:** No file header
10	FLAM_NOT_FLAMFILE **Meaning:** File is not a FLAMFILE

Return Code	Symbolic Name - Meaning
11	FLAM_FLAMFILE_FORMAT_ERROR **Meaning:** FLAMFILE formatting error
12	FLAM_FLAMFILE_RECORD_SIZE **Meaning:** Invalid record size for compressed file
13	FLAM_FILE_LENGTH **Meaning:** FLAMFILE incomplete or damaged
14	FLAM_CHECKSUM **Meaning:** Checksum error
15	FLAM_RECORD_GR_32KB **Meaning:** Original record larger than 32764 bytes
16	FLAM_RECORD_GR_BUFFER **Meaning:** Original record larger than matrix - 4
17	FLAM_FLAM_VERSION_1 **Meaning:** FLAM compressed file is Version 1
20	FLAM_ILLEGAL_FLAM_OPEN **Meaning:** Open mode illegal
22	FLAM_ILLEGAL_COMPRESSION_MODE **Meaning:** Compression mode illegal
25	FLAM_ILLEGAL_RECORD_SIZE **Meaning:** Record size illegal
26	FLAM_ILLEGAL_CHARACTER_SET **Meaning:** Character set illegal

Return Code	Symbolic Name - Meaning
29	FLAM_ERR_PW **Meaning:** Password wrong or missing
30	FLAM_FILE_EMPTY **Meaning:** Input file empty
31	FLAM_FILE_NOT_FOUND **Meaning:** Input file does not exist
32	FLAM_OPEN_MODE **Meaning:** Invalid open mode
34	FLAM_RECORD_FORMAT **Meaning:** Invalid record format
35	FLAM_RECORD_SIZE **Meaning:** Invalid record size
39	FLAM_NO_VALID_FILENAME **Meaning:** File name not valid
43	FLAM_ERR_EXIT **Meaning:** Abnormal termination by exit routine
46	FLAM_OPEN_MSGFILE **Meaning:** Cannot open message file
47	FLAM_ERR_OUTPATH **Meaning:** -outpath only allowed with [ ], [*], or [filename]
53	FLAM_PATH_INVALID **Meaning:** Environment variable FLAM_PATH not defined

Return Code	Symbolic Name - Meaning
56	FLAM_FILEHEADER_GR_32KB **Meaning:** File header of compressed file longer than 32 KB
57	FLAM_KOMP_LENGTH **Meaning:** Compressed file length invalid
59	FLAM_ERR_NONDC **Meaning:** ndc only allowed with adc, aes, or flamenc
60	FLAM_SYNTAX **Meaning:** Error in compressed file
65	FLAM_CONS_RECORD **Meaning:** Consistency point invalid
70	FLAM_NOT_VALID_VERSION **Meaning:** FLAM compressed file is not Version 2x
71	FLAM_FL_CUT **Meaning:** Decompressed record cut (flamup error)
72	FLAM_RECORD_CUT **Meaning:** Decompressed record(s) cut (warning from flamup)
73	FLAM_ERR_KOMP_LENGTH **Meaning:** Wrong length of compressed data
74	FLAM_CHECK_CHARACTER **Meaning:** Check character error
81	FLAM_PARAM_ERR **Meaning:** Unknown parameter/qualifier

Return Code	Symbolic Name - Meaning
82	FLAM_PARAMETER_VALUE **Meaning:** Parameter value invalid
85	FLAM_SAME_NAME **Meaning:** Same name for input and output files
87	FLAM_NO_INPUT_FILE **Meaning:** Input file has no name
88	FLAM_NO_OUTPUT_FILE **Meaning:** Output file has no name
90	FLAM_QUALIFIER_VALUE **Meaning:** Qualifier value invalid
92	FLAM_NO_REPLACING_CHARACTERS **Meaning:** Cannot substitute any characters in input file name
98	FLAM_READ_ERROR **Meaning:** Error occurred while reading a record
99	FLAM_WRITE_ERROR **Meaning:** Error occurred while writing a record
101	FLAM_NOT_ALL_INPUT_FILES **Meaning:** Some input files not processed.
102	FLAM_ANZ_OUTFILE_GR_INFILE **Meaning:** Number of output files greater than number of input files
271	FLAM_ERR_BAD_KMEXIT **Meaning:** Wrong specification of key management exit

Return Code	Symbolic Name - Meaning
272	FLAM_ERR_KMEX1 **Meaning:** Error loading the key management exit
351	FLAM_SEC_ERR_351 **Meaning:** SECURITY: Member number not contiguous
352	FLAM_SEC_ERR_352 **Meaning:** SECURITY: Member trailer: counters wrong
354	AES_MEM_MAC **Meaning:** AES: Member MACs wrong
355	AES_FIL_MAC **Meaning:** AES: File MACs wrong
356	AES_CRC_SUFF **Meaning:** Bad checksum in compressed data (CRC-Offs)
357	AES_CRC_5 **Meaning:** Bad checksum in compressed data (CRC-5)
360	FLAM_ERR_REDUNDANT **Meaning:** Parameter redundant or ambiguous
401	FLAM_SEQ_OPD **Meaning:** Wrong sequence in flmopd call.
402	FLAM_SEQ_OPF **Meaning:** Wrong sequence in flmopf call.
403	FLAM_SEQ_PHD **Meaning:** Wrong sequence in flmphd call.

Return Code	Symbolic Name - Meaning
404	FLAM_SEQ_GHD **Meaning:** Wrong sequence in flmghd call.
405	FLAM_SEQ_PUH **Meaning:** Wrong sequence in flmpuh call.
406	FLAM_SEQ_GUH **Meaning:** Wrong sequence in flmguh call.
407	FLAM_SEQ_PUT **Meaning:** Wrong sequence in flmput call.
408	FLAM_SEQ_GET **Meaning:** Wrong sequence in flmget call.
409	FLAM_SEQ_POS **Meaning:** Wrong sequence in flmpos call.
410	FLAM_SEQ_FLU **Meaning:** Wrong sequence in flmflu call.
411	FLAM_SEQ_CLS **Meaning:** Wrong sequence in flmcls call.
412	FLAM_SEQ_PWD **Meaning:** Wrong sequence in flmpwd call.
900	FLAM_PARAMETER_QUALIFIER **Meaning:** FLAM options invalid (flamup only)
998	FLAM_INVALID_LICENSE **Meaning:** License invalid

Return Code	Symbolic Name - Meaning
999	FLAM_INVALID_FUNCTION **Meaning:** Required call sequence was violated or request for memory could not be met

I/O error flags in the most significant byte of the return code:

1st half-byte	Flag	Affected files
	X'ex'	Input file
	X'ax'	Output file
	X'fx'	FLAMFILE
	X'cx'	Parameter file
	X'dx'	Message filei
	X'9x'	Code table
	X'8x'	Default values
	X'7x'	Messages

2nd half-byte	Flag	Origin of error
	X'x0'	FLAM message
	X'xf'	I/O function message

FLAM (UNIX)

User Manual

Appendix B:

Code Tables

B. Code tables

If you need to exchange files between systems which use different character sets, you can ask FLAM to convert your data to the desired codes automatically by specifying the parameter translate=code-table. You must make a translation table available to FLAM in order to do so.

This chapter describes how to generate a code table.

B.1 Structure of code tables

FLAM reads the first 256 bytes of the file you have specified as a code table. These 256 bytes form a string, which is used by FLAM as a translation table. The numeric value of the code for each character of the original data is taken as an index for the translation table and replaced by the corresponding table element. The possible value range of this numeric value is 0 - 255 (decimal), which corresponds to table positions 1 - 256.

The following code table can be generated, for example, in order to convert EBCDIC-coded text data to ASCII code for the decompression procedure:

Example B.1: Contents of a code table

```
********************************
********************************
 **********.*(***********!**);*
_**********,***?**********:**'*"
*abcdefghi*******jklmnopqr******
**stuvwxyz**********************
*ABCDEFGHI*******JKLMNOPQR******
**STUVWXYZ******0123456789******
```

This table can be used to translate EBCDIC data containing alphanumeric characters and the following special characters: . (!) ; - , ? : ' ". All other EBCDIC codes are converted to the ASCII character *.

The position of each character in the translation table is derived by adding 1 to the numeric value of its EBCDIC code. The number 9, for example, corresponds to the hexadecimal EBCDIC code F9 (decimal 249) and its position is 250. If, for instance, the German umlaut ü needed to be added to the table, its EBCDIC code would have to be determined first. Assuming this code to be hexadecimal D0 (decimal 208), the ASCII code for * would have to be replaced by the ASCII code for ü at position 209 (directly in front of the A).

Conversely, the EBCDIC code F9 (hexadecimal) would have to be entered at position 58 for the number 9 (ASCII code hexadecimal 39 = decimal 57) in a table for translating ASCII to EBCDIC.

B.2 Generating code tables

When code tables are generated, it is not normally possible to enter character codes which correspond to non-printing ASCII characters using the keyboard. It is advisable to use the following program, which incorporates a modified code table (see example in /opt/limes/flam/doc/examples/cr_code.c):

Example B.2: ASCII - EBCDIC translation table

```
/*
Example:
ASCII - EBCDIC translation table
*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <fcntl.h
extern int errno;
extern int sys_nerr;

main()

{

int status;
unsigned long fk;
static char filename[20] = {"code_ae.dat"};

static char ebcdic[256] = {0x00, 0x01, 0x02, 0x03, 0x37, 0x2d, 0x2e, 0x2f,
                           0x16, 0x05, 0x25, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
                           0x10, 0x11, 0x12, 0x13, 0x3c, 0x3d, 0x32, 0x26,
                           0x18, 0x19, 0x3f, 0x27, 0x1c, 0x1d, 0x1e, 0x1f,
                           0x40, 0x5a, 0x7f, 0x7b, 0x5b, 0x6c, 0x50, 0x7d,
                           0x4d, 0x5d, 0x5c, 0x4e, 0x6b, 0x60, 0x4b, 0x61,
                           0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7,
                           0xf8, 0xf9, 0x7a, 0x5e, 0x4c, 0x7e, 0x6e, 0x6f,
                           0x7c, 0xc1, 0xc2, 0xc3, 0xc4, 0xc5, 0xc6, 0xc7,
                           0xc8, 0xc9, 0xd1, 0xd2, 0xd3, 0xd4, 0xd5, 0xd6,
                           0xd7, 0xd8, 0xd9, 0xe2, 0xe3, 0xe4, 0xe5, 0xe6,
                           0xe7, 0xe8, 0xe9, 0xbb, 0xbc, 0xbd, 0x6a, 0x6d,
                           0x4a, 0x81, 0x82, 0x83, 0x84, 0x85, 0x86, 0x87,
                           0x88, 0x89, 0x91, 0x92, 0x93, 0x94, 0x95, 0x96,
                           0x97, 0x98, 0x99, 0xa2, 0xa3, 0xa4, 0xa5, 0xa6,
                           0xa7, 0xa8, 0xa9, 0xfb, 0x4f, 0xfd, 0xff, 0x07,
                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                           0x00, 0x00, 0x00, 0x00, 0xff, 0x06, 0x01, 0x00
                          };
```

```
fk = open(filename, O_CREAT | O_WRONLY | O_TRUNC);
if (fk == -1)
  { perror("Open file");
    exit(1);
  };
status = chmod(filename, S_IWUSR | S_IRUSR| S_IRGRP | S_IROTH);
if (status == -1)
  { perror("chmod");
    exit(1);
  };

status = write(fk, ebcdic, 256);
if (status != 256)
  perror("Write error");

status = close(fk);

exit(0);

}
```

