FLSSRV - User Manual

limes datentechnik gmbh

Version 5.1.20, May 10 2019

Table of Contents

released

Frankenstein Limes Integrated Extended Security (FLIES®)

Trademarks

Below you can find all trademarks or registered trademarks of limes datentechnik ® gmbh. These trademarked terms are marked below with the appropriate symbol (® or ™), indicating registered or common law trademarks owned by limes datentechnik ® gmbh at the time this information was published. The following terms are trademarks of the limes datentechnik ® gmbh in Germany other countries, or both:

- limes datentechnik® - Company name of the owner of this document
- FLCL® - Frankenstein Limes Command Line
- FLCC® - Frankenstein Limes Control Center
- FLAM® - Frankenstein Limes Access Method
- FLUC® - Frankenstein Limes Universal Converter
- FLIES® - Frankenstein Limes Integrated Extended Security
- FLAMFILE® - A file based on FLAM syntax

Abstract

This document provides information about using the Frankenstein Limes Integrated Extended Security (FLIES) Server. It contains advanced guidelines and information to run this network server. With the FLIES server FLAMFILES can be accessed over the net.

PREFACE

The FLSSRV command line is implemented with the CLE/P library. The CLE/P library was developed by limes datentechnik and released as open source under the ZLIB license. CLE/P is a compiler to provide a platform independent command line interface for each kind of batch processing environment. The CLE/P library provides a lot of features including all the built-in functions below. For example: We use this library to automatically create this document as part of our build process; If we add a parameter to the CLE/P tables and build the FL5 project, this manual will be regenerated as well in order to be always up to date.

This manual is generated with the built-in function GENDOCU provided by the CLE/P library by calling the command below:

```
flssrv gendocu flssrvbk.txt
```

Based on that the document is always in sync with the implementation but some redundancies are the result of such generated documentation. This manual was designed as a reference book and in this context it is useful for a parameter to be described in all places where it is of relevance.

Chapter 1. COMMAND LINE PROCESSOR

This command line parser provides a list of commands and some built-in functions, please use *MANPAGE*, *HELP* and *SYNTAX* to get extensive information about these capabilities.

Additionally the build-in function *GENDOCU* can generate a complete or a part of the user manual.

To read the parameter of a command, a compiler (command line processor CLP) is applied. To see the regular expressions (lexems) and the corresponding grammar, please use the built-in functions *LEXEM* and *GRAMMAR*.

The return/condition/exit codes of the executable and the reason codes of the different commands can be reviewed with the built-in function *ERRORS*. See RETURN CODES and, if available, REASON CODES for the meaning of the used return and reason codes.

The command line executer (CLE) uses an owner management in order to separate the settings for different clients and a property management for each command. If problems occur, a trace might be activated.

For each command execution the owner and environment variables can be defined in the configuration file.

The default trace file is *stdout*. If the trace is activated before a trace file is defined, the trace will be printed on the screen.

Last but not least it is possible to view the license, version and other information about the program.

This is the RELEASE build version of the FLAMCLEP.

1.1. USED ENVIRONMENT VARIABLES

- LANG - to determine the CCSID on EBCDIC systems
- HOME - to determine the home directory on UNIX/WIN
- USER - to determine the current user id on UNIX/WIN
- ENVID - for the current environment qualifier (D/T/P) if key label template mapping used (default is *T*)
- OWNERID - used for current owner if not already defined
- CLP_NOW - The current point in time used for predefined constants (0tYYYY/MM/DD.HH:MM:SS)
- CLP_STRING_CCSID - CCSID used for interpretation of critical punctuation character on EBCDIC systems (default is taken from LANG)
- FLSSRV_CONFIG_FILE - the configuration filename (default is *$HOME/.flssrv.config* on UNIX/WIN or *&SYSUID..FLSSRV.CONFIG* on mainframes)
- FLSSRV_DEFAULT_OWNER_ID - the default owner ID (default is *limes*)
- owner_FLSSRV_command_PROPERTY_FILENAME - To override default property file names
- path_argument - to override the hard coded default property value

Environment variables can be set from the system or in the program configuration file with the build-in function *SETENV*. Particularly on mainframe systems the configuration file is an easy way to define

environment variables. Additionally on host systems the DD name STDENV and with lower priority the data set name *&SYSUID..STDENV* is supported. On the other architectures the file *.stdenv* in the working or with lower priority in the home directory can be used to define environment variables for FLAM utility, subsystems and subprogram interfaces.

The STDENV files allows to define the FLSSRV_CONFIG_FILE, FLSSRV_DEFAULT_OWNER_ID, LANG and other environment variables in JCL:

Below a you can find a sample with a z/OS DD name allocation.

```
//STDENV   DD *
FLSSRV_CONFIG_FILE=GLOBAL.FLSSRV.CONFIG
LANG=de_DE.IBM-1141
HOME=/u/hugo
USER=hugo
ENVID=T
/*
```

The definition of the LANG variable outside of the program configuration file is recommended, so that the system character set is defined in each situation. The program configuration file name is used as is, there are no replacements (<USER>) possible, because it is used in front of establishment of the environment it self.

Often it will be useful to have a dedicated environment per user on mainframes. In such a case it makes sense to define the environment in a dedicated file for each user.

```
//STDENV   DD DSN=USER.ENVIR(&SYSUID.), DSP=SHR
```

Since z/OSv2r1 you can also used exported JCL symbols like environment variables. The exported JCL symbols have lower priority then the same environment variable. The service CEEGTJS is called dynamically and the language environment must be in the STEPLIB concatenation to use it. Additional all z/OS system symbols can be used in the in the string replacement (<&LDAY>). The environment variable have the highest priority, followed by the exported JCL symbols. The lowest priority have system symbols. If the variable name not found, then no replacement are done and the angle bracket keep still in place.

```
//E0       EXPORT SYMLIST=*
//S1       SET    BLKSIZE=27886
//S3       SET    LRECL=128
```

Beside all the environment variables managed by CLE it is possible to set all properties as environment variables to override the hard coded default values with CLP. If a property is defined as environment variable and in a property file, then the value in the property file overrides the setting in the environment. The environment variable name for each property is build by the rules below:

- convert all letters to upper case
- replace all dots (.) by underline (_)

To get a list and help for all properties please use the built-in function *GENPROP* to generate property files. The properties can be defined per owner, per program and general. The owner specific definition overrides the program specific definition and the program specific definition overrides the general

definition. Examples:

```
CONV_READ_TEXT_ENL2LF=OFF #in general the 0x15 to 0x25 conversion is off#
HUGO_FLCL_CONV_READ_TEXT_ENL2LF=ON # for owner 'hugo' the conversion is on#
```

The value string behind the sign (including the comment) will be used as supplement for the command line processor. Aliases are not supported in this case. You can only define properties for the main argument. If a string must be enclosed with apostrophe, please don't use double quotation marks, because these are used in addition if a new property file is build based on the environment settings.

```
FLCL_ICNV_FROM='IBM-1141'         # this is the best solution
FLCL_ICNV_TO=UTF-8                # "UTF-8" could result in errors
```

See PROPERTIES for the current property file content.

1.2. ENVIRONMENT VARIABLE MAPPING

Each value enclosed with angle brackets (<>) are replaced with the corresponding environment variable (<LANG>). This feature is available at each starting point of a lexem and inside of a normal string.

If the environment variable not defined the replacements below are still possible: * <SYSUID> - Current user id in upper case * <USER> - Current user id (case sensitive) * <CUSER> - Current user id in upper case == <SYSUID> * <Cuser> - Current user id in title case * <cuser> - Current user id in lower case * <HOME> - Replaces with the users data directory, if this not available the replacements below are done: **On UNIX with /home/<USER>** On USS with /u/<USER> ** On ZOS with <SYSUID>

1.3. FILENAME MAPPING

All filenames used by CLEP are additionally mapped based on the rules below:

- The tilde character (~) is replaced with the string "<HOME>"
- DD names on mainframes must be prefixed with "DD:"
- Data set names on mainframes are always full qualified
- Path names on mainframes must contain a least one slash (/)
- Data set names on USS must start with //
  - Full qualified names with HLQ must enclose in apostrophes ("//'")
  - If apostrophes are not used the SYSUID is prefixed as HLQ
- Normal file names on other platforms could be relative

ATTENTION: If a requested environment variable is not defined, the replacement is done with the empty string. This can result in unexpected behavior.

To use a "<" or "~" as a part of a filename the character must be specified twice.

Beside this rules the replacement technologies of your shell can be used, but on some platforms $HOME, $USER or something similar might not be available, for such cases the possibilities above are implemented.

This file name mapping is provided by the library CLEPUTL and should also be used for file names managed by the commands supported with this program.

1.4. KEY LABEL NAME MAPPING

The key label name mapping works like the file name mapping, but you can additional use `^` to replace it with the OWNERID (`^=<ONWERID>`), `!` to replace it with the environment identifier (`!=<ENVID>`) and `~` is only replaced with login user id. If the environment ID not defined then `T` for test is used by default.

1.5. SPECIAL EBCDIC CODE PAGE SUPPORT

To interpret commands correctly the punctuation characters below are on different code points depending on the EBCDIC CCSID used to enter these values.

```
CRITICAL PUNCTUATION CHARACTERS: ! $ # @ [ \ ] ^ ` { | } ~
```

These critical characters are interpreted normally dependent on the environment variable LANG. If the environment variable LANG is not defined then the default CCSID IBM1047 (Open Systems Latin-1) is used. Below is the current list of supported CCSIDs on EBCDIC systems.

```
SUPPORTED EBCDIC CODE PAGES FOR COMMAND ENTRY:
   "IBM-1140","IBM-1141","IBM-1142","IBM-1143",
   "IBM-1144","IBM-1145","IBM-1146","IBM-1147",
   "IBM-1148","IBM-1149","IBM-1153","IBM-1154",
   "IBM-1156","IBM-1122","IBM-1047","IBM-924",
   "IBM-500","IBM-273","IBM-037","IBM-875","IBM-424",
   "IBM-277","IBM-278","IBM-280","IBM-284","IBM-285",
   "IBM-297","IBM-871","IBM-870","IBM-1025","IBM-1112",
   "IBM-1157"
```

You can define the code page explicitly (LANG=de_DE.IBM-1141) or only the language code (LANG=de_DE, LANG=C). If only the language code defined then the CCSID is derived from the language code (DE=IBM-1141, US=IBM-1140, C=IBM-1047, ...).

If possible, these critical characters are also converted for print outs. At output it is not possible to convert anything correctly, because some strings for print out are coming from other sources (like system messages and others). Only all known literals are converted, for unknown variables such a conversion is not possible and CLEP expects that such strings are encoded in the correct system code page, but there is no guaranty for this.

To be independent of the environment escaping of the CLP strings is possible for the critical

punctuation characters on EBCDIC systems. See list below:

```
! = &EXC;   - Exclamation mark
$ = &DLR;   - Dollar sign
# = &HSH;   - Hashtag (number sign)
@ = &ATS;   - At sign
[ = &SBO;   - Square bracket open
\ = &BSL;   - Backslash
] = &SBC;   - Square bracket close
^ = &CRT;   - Caret (circumflex)
` = &GRV;   - Grave accent
{ = &CBO;   - Curly bracket open
| = &VBR;   - Vertical bar
} = &CBC;   - Curly bracket close
~ = &TLD;   - Tilde
```

A escape sequence starts with the ampersand (&) followed be 3 character (not case sensitive) and is terminated with semicolon (;). With an X and 2 hexadecimal digit the definition of a binary byte value are possible. If such a sequence in the string required then the ampersand must be typed twice (&&). To mark a part or the of whole string in a certain CCSID the escape sequence must contain 1 to 6 digits. Few samples can be found below:

```
&1047; all in 1047 &0; reset to system code page &1140; no in 1140
```

At the beginning the system code page is active except the environment variable CLP_STRING_CCSID is defined. The defined CCSID is valid till the next CCSID definition. An unsupported CCSID (0) can be used to restore the interpretation to the system code page.

ATTENTION: This is no character set conversion, only the UNICODE code points 0-127 which are on different code points in the different EBCDIC code pages are replaced. All other higher code points (e.g. German Umlauts (*ä*,*ü*,*Ö*)) are not touched.

The partial CCSID conversion are mainly useful for application programming interfaces. At compile time the CCSID for literals must be defined. This CCSID could be differ from the system CCSID (local character set) of variable parameter. In such a case a application can mark the literal part in the CCSID used for literals at compile time, and the variable part could be conform to the CCSID defined over the LANG variable. See the C example below:

```
snprintf(acCmd,sizeof(acCmd),"&1047;get.file='&0;%s&1047'",pcFilename);
```

C-Code are normally in 1047 and if no literal conversion defined then the literals also in 1047. The file name is a parameter in local character set. In Cobol the default code page for literals is 1140. To define the CCSID for CLP strings from outside, the environment variable CLP_STRING_CCSID can be defined. This is useful if an application cannot recompile with the correct code page for the literals.

On ASCII/UTF-8 platforms a miss-interpretation of punctuation characters smaller than 128 is not possible. On such platforms the LANG variable is not used for command interpretation or printouts. The escape and CCSID sequences are supported but have no effect on these platforms. This helps to develop platform independent code for CLP strings.

Chapter 2. PROGRAM *flssrv*

2.1. SYNOPSIS

```
HELP:   FLIES Server (Remote FLAM)
PATH:   limes
TYPE:   PROGRAM
SYNTAX: :> flssrv COMMAND/FUNCTION ...
```

2.2. DESCRIPTION

The FLIES® server allows remote read and write access to a FLAMARCHIVE (file, database, VSAM) by using a network connection with one of the supported protocols (IP, MQ, ...). Additionally it can act as a gateway to forward the client requests to another server. Search requests for segments are also possible, thereby making the limes access method network transparent. The segements transferred on the network are compressed and encrypted by the client if this was requested. This keeps control on the client side and reduces the amount of network traffic.

2.2.1. USED ENVIRONMENT VARIABLES

- LANG - to determine the default CCSID for this platform (Format: de_DE.UTF-8)
- FLSSRV_CONFIG_FILE - defines the config filename (if not set default is *.flssrv.config*)
- FLSSRV_DEFAULT_OWNER_ID - defines the default owner ID (if not set default is *limes*)

The enviroment variables can be set over the system or over the FLSSRV config file with build-in function *SETENV*. Especially on mainframe systems the config file is a easy way to define environment variables for FLSSRV.

FLIES - Frankenstein Limes Integrated Extended Security

2.3. SYNTAX

Below you can see the main syntax for this program. This content is printed on the screen if the built-in function *SYNTAX* is used.

Each command can be entered as an argument list on the command line or in a parameter file. To define a parameter file the assignment character (=) must be used. For an argument list a blank or if the command (main table) is of type overlay a dot (.) or if the command is of type object a round bracket open (() must be given. If the round bracket open is used then the corresponding round bracket close ()) at the end of the command is mandatory. The dot and parenthesis are in accordance with the normal syntax.

If an argument list is used all parameters (argv[]) are concatenated to one long string and this string is passed to the parser, to allow usage of all features of the shell to build the command line input. To make sure that anything you enter is passed one by one to the parser please use double (WINDOWS®, UNIX) or single (z/OS®) quotation marks at the beginning and end of the argument list. If the dot for overlays or parenthesis for objects are used then the complete command must be included in double quotation marks. The examples below show all possible command entries for the type object:

```
:> FLSSRV command temp() dummy='str'
:> FLSSRV command(temp() dummy='str')
:> FLSSRV command "temp() dummy='str'"
:> FLSSRV "command(temp() dummy='str')"
```

For a command of type overlay it looks like:

```
:> FLSSRV command temp()
:> FLSSRV command.temp()
:> FLSSRV command "temp()"
:> FLSSRV "command.temp()"
```

In a parameter file you can start with a dot for an overlay or with parenthesis for an object or with the argument (overlay) or argument list (object) directly.

```
Syntax for program 'flssrv':
--| Commands: RUN
--|--| flssrv [OWNER=oid] command "... argument list ..." [MAXCC=[max][-min]]
[QUIET/SILENT]
--|--| flssrv [OWNER=oid] command=" parameter file name " [MAXCC=[max][-min]]
[QUIET/SILENT]
--|--| You can optionally specify:
--|--|--| the owner id for this command (to use custom configuration files)
--|--|--| the maximum condition code (max) to suppress warnings
--|--|--| the minimum condition code (min), zero is returned if the condition code
would be smaller
--|--|--| QUIET disables the normal log output of the command line executer
--|--|--| SILENT disables log and errors messages of the command line executer
--| Built-in functions:
--|--| flssrv SYNTAX [command[.path] [DEPTH1 | ... | DEPTH9 | ALL]]
--|--| flssrv HELP [command[.path] [DEPTH1 | ... | DEPTH9 | ALL]] [MAN]
--|--| flssrv MANPAGE [function | command[.path][=filename]] | [filename]
--|--| flssrv GENDOCU [command[.path]=]filename [NONBR]
--|--| flssrv GENPROP [command=]filename
--|--| flssrv SETPROP [command=]filename
--|--| flssrv CHGPROP command [path[=value]]*
--|--| flssrv DELPROP [command]
--|--| flssrv GETPROP [command[.path] [DEPTH1 | ... | DEPTH9 | DEPALL | DEFALL]]
--|--| flssrv SETOWNER name
--|--| flssrv GETOWNER
--|--| flssrv SETENV variable=value
--|--| flssrv GETENV
--|--| flssrv DELENV variable
--|--| flssrv TRACE ON | OFF | FILE=filename
--|--| flssrv CONFIG [CLEAR]
--|--| flssrv GRAMMAR
--|--| flssrv LEXEM
--|--| flssrv LICENSE
--|--| flssrv VERSION
--|--| flssrv ABOUT
--|--| flssrv ERRORS
```

2.4. HELP

Here you can see the static main help for this program. This content is printed on the screen if the built-in function *HELP* is used without any further arguments.

```
Help for program 'flssrv':
--| Commands - to execute powerful subprograms
--|--| flssrv RUN      - run server
--| Built-in functions - to give interactive support for the commands above
--|--| flssrv SYNTAX   - Provides the syntax for each command
--|--| flssrv HELP     - Provides quick help for arguments
--|--| flssrv MANPAGE  - Provides manual pages (detailed help)
--|--| flssrv GENDOCU  - Generates auxiliary documentation
--|--| flssrv GENPROP  - Generates a property file
--|--| flssrv SETPROP  - Activate a property file
--|--| flssrv CHGPROP  - Change a property value in the currently active property file
--|--| flssrv DELPROP  - Remove a property file from configuration
--|--| flssrv GETPROP  - Show current properties
--|--| flssrv SETOWNER - Defines the current owner
--|--| flssrv GETOWNER - Show current owner setting
--|--| flssrv SETENV   - Set an environment variable
--|--| flssrv GETENV   - Show the environment variables
--|--| flssrv DELENV   - Delete an environment variable
--|--| flssrv TRACE    - Manage trace capabilities
--|--| flssrv CONFIG   - Shows or clear all the current configuration settings
--|--| flssrv GRAMMAR  - Shows the grammar for commands and properties
--|--| flssrv LEXEM    - Shows the regular expressions accepted in a command
--|--| flssrv LICENSE  - List license information for the program
--|--| flssrv VERSION  - List version information for the program
--|--| flssrv ABOUT    - Show information about the program
--|--| flssrv ERRORS   - Show information about return and reason codes of the program
For more information please use the built-in function 'MANPAGE'
```

Chapter 3. Available commands

Commands are used to run powerful subprograms. The command line processor compiles a table defined syntax in a corresponding preinitialized data structure. This structure will be mapped to the parameter structure of the corresponding subprogram. The subprogram can be executed with the defined arguments/parameters.

To support these variable commands several built-in functions are available and described in the next section.

Each supported command is explained in a separate section in this document. Each section contains a synopsis including the help message, the path, the type and the syntax followed by a detailed description.

If an argument of this command is an object or overlay or, if a detailed description is available for this argument, a separate section with synopsis (help, path, type, syntax) and description is written, otherwise a bullet list is printed which contains the keyword, the syntax and the help message.

For the syntax of an overlay braces {} are used to keep all possible arguments of an overlay logically together. These braces are not part of the real syntax. The braces are only written to demonstrate clearly that one of these arguments must be selected with the DOT operator (optional) for defining the overlay.

To be compatible with certain shells, the features below are implemented:

- Strings can be enclosed with single '' or double "" quotation marks
- Integrated strings (without spaces) can also defined without quotes
- Keywords can also start with "-" or "--" in front of the qualifier
- If it unique then parenthesis and the dot can be omit for object and overlays

Commands can be declared in deep hierarchical depth. In order to simplify their handling the path is a powerful instrument for managing only relevant parts of it. Because of that the path is printed for each synopsis.

To run commands under different owners, the owner id can be defined in front of the command.

To suppress the outputs of the command line executer you can set QUIET or SILENT as the last parameter. Additional to QUIET, SILENT suppressed also the error messages. These options has no effect of the outputs of the commands and build-in functions.

For job control, the maximum (and minimum) condition code (MAXCC=[max][-min]) of the command execution can optionally be set as the last parameter for each command. The value *max* is the maximum condition code. If the condition code, that would be returned if MAXCC was not specified, would be greater than *max*, it is reduced to the value of *max*. If the condition code would be smaller than *min*, the actually returned condition code is reduced to 0. Both, *max* and *min*, values are optional. To only specify the minimum, simply omit the *max* value, i.e. the MAXCC value starts with a hyphen. Example: The clause "MAXCC=8-4" will set each condition code greater than 8 to 8 and each condition code smaller than 4 to 0, while "MAXCC=-4" only sets condition codes smaller than 4 to 0.

```
flssrv [OWNER=oid] command "... argument list ..." [MAXCC=[max][-min]] [QUIET]
flssrv [OWNER=oid] command=" parameter file name " [MAXCC=[max][-min]] [QUIET]
```

The parameter for each command can be provided as argument list on the command line or as

parameter file.

3.1. COMMAND *RUN*

SYNOPSIS

```
HELP:   run server
PATH:   limes.flssrv
TYPE:   OBJECT
SYNTAX: :> flssrv
RUN(PORT='str',CLIENT(),ROOTPATH='str',HLEN=num,DLEN=num,HANDLE_CLIENT=THREAD/PROCESS,
DAEMONIZE,LOG())
```

DESCRIPTION

The RUN command must be used to to execute the FLIES® server. It is an object containg all parameters needed to configure its behaviour.

To run the server with default values

```
   FLSSRV RUN
```

is sufficient. To get help for a parameter use:

```
   FLSSRV HELP RUN.parameter[.parameter[...]]
```

To read the manual page for a parameter, please use:

```
   FLSSRV MANPAGE RUN.parameter[.parameter[...]]
```

To generate the user manual for the RUN command use:

```
   FLSSRV GENDOCU RUN=filename
```

Parameters can be defined by the command line (directly or per file) or by properties read from the corresponding property file.

EXAMPLES

```
   FLSSRV RUN rootpath='/srv/flam'
   FLSSRV RUN handle_client=process
```

ARGUMENTS

- `NUMBER: HLEN=num - Initial buffer length for header`
- `NUMBER: DLEN=num - Initial buffer length for data`

3.1.1. PARAMETER *PORT*

SYNOPSIS

```
HELP:   Port number [17996]
PATH:   RUN
TYPE:   STRING
SYNTAX: PORT='str'
```

DESCRIPTION

The PORT parameter allows to set the IP port number the server will listen on for incoming connection requests. If not specified the default port 17996 is used.

NOTE The port number has to be specified as a string.

EXAMPLES

```
   FLSSRV RUN port='3192'
```

3.1.2. OBJECT *CLIENT*

SYNOPSIS

```
HELP:   Client IPN Parameter
PATH:   RUN
TYPE:   OBJECT
SYNTAX: CLIENT(HOST='str',PORT='str')
```

DESCRIPTION

The CLIENT parameter determines where to forward incoming requests to.

If specified, the server acts as a gateway and will forward all requests received from a client to another server and all answers from this other server are send to the requesting client. This is transparent for the client. The CLIENT parameter is an object with 2 arguments: HOST and PORT. The HOST argument is required and is the IP addres of the remote system where another FLIES Server is running. The PORT argument defines the IP port number used by the remote server to listen for incoming connect requests. If not spefified 17996 is used.

NOTE The host and port arguments must be specified as strings.

EXAMPLES

```
FLSSRV RUN client(host='192.168.178.34')
```

ARGUMENTS

- STRING: HOST='str' - IP address [localhost]
- STRING: PORT='str' - Port number [17996]

3.1.3. PARAMETER *ROOTPATH*

SYNOPSIS

```
HELP:   Root path to use for local files
PATH:   RUN
TYPE:   STRING
SYNTAX: ROOTPATH='str'
```

DESCRIPTION

The ROOTPATH parameter allows to set the root directory for the server. The directory specification in all requests received from a client will be regarded relative to this root directory. If omitted the root directory will be set to the current working directory.

EXAMPLES

```
FLSSRV RUN rootpath='/srv/flam'
```

3.1.4. PARAMETER *HANDLE_CLIENT*

SYNOPSIS

```
HELP:   How to handle client connection [THREAD]
PATH:   RUN
TYPE:   NUMBER
SYNTAX: HANDLE_CLIENT=THREAD/PROCESS
```

DESCRIPTION

The HANDLE_CLIENT parameter allows to define how the server will handle a client connection.

With each successful connect of a client, the server ether starts a new thread or a new process to handle the requests of this client. Usually a thread is started more quickly than a process, but will run in the same address space as other threads of the same process. Handling client connections in processes needs more resources but is more secure than threads. The decision how to handle client connections is dependent on the security requirements and the average traffic load the server must be able to handle.

EXAMPLES

```
FLSSRV RUN handle_client=process
```

SELECTIONS

- `THREAD - handle client connection in thread`
- `PROCESS - handle client connection in process`

3.1.5. PARAMETER *DAEMONIZE*

SYNOPSIS

```
HELP:   run server in background [OFF]
PATH:   RUN
TYPE:   SWITCH
SYNTAX: DAEMONIZE
```

DESCRIPTION

The DAEMONIZE parameter will cause the server to run as a UNIX daemon.

EXAMPLES

```
FLSSRV RUN daemonize
```

On WINDOWS this parameter has no effect. To run the FLIES server as a WINDOWS service 2 utility programs are provided: flsrun and flscontrol

To install the FLIES server as a WINDOWS service

```
flsrun install
```

must be executed once with system administrator privileges. The service name of the FLIES server is FLM

Control of this service can be done from the command line with

```
    flscontrol enable FLM
    flscontrol disable FLM
    flscontrol start FLM
    flscontrol stop FLM
    flscontrol delete FLM
```

to start/stop, enable/disable or deleting the service. This control is, of course also possible with the WINDOWS service manager.

3.1.6. OBJECT *LOG*

SYNOPSIS

```
HELP:   Logging Parameter
PATH:   RUN
TYPE:   OBJECT
SYNTAX: LOG(MESSAGE(),DESTINATION.{})
```

DESCRIPTION

The LOG parameter allows flexible adjustment for the logging of messages. It is possible to specify the destination as well as which kind of message will be written.

For more information please have a look at the man pages for each parameter of LOG.

3.1.7. OBJECT *MESSAGE*

SYNOPSIS

```
HELP:   Select type of messages to log [ALL]
PATH:   RUN.LOG
TYPE:   OBJECT
SYNTAX:
MESSAGE(NONE,ERROR,ERRTRACE,WARNING,NOTICE,DEBUG,ABOUT,VERSION,INPUT,PARAMETER,STATIST
IC,SUMMARY,INFO,LICID,LICENSE,PROGRESS,USAGE,CONDBG,ALL,DEFAULT,MINIMAL,ERRONLY)
```

DESCRIPTION

The log message specification is a classification of the messages to write to the selected log destination. Each type is selectable individually by adding its argument inside this object.

For added convenience, the special types NON, ALL, MINIMAL select the proper message types with just one argument.

If no message specification is given, a DEFAULT is used which writes out messages of these types:

- ERROR
- ERRTRACE
- WARNING
- NOTICE
- PARAMETER
- STATISTIC
- INFO
- LICID

ARGUMENTS

- `SWITCH: NONE - No message will be logged`
- `SWITCH: ERROR - Error message will be logged`
- `SWITCH: ERRTRACE - Error trace will be logged`
- `SWITCH: WARNING - Warning will be logged`
- `SWITCH: NOTICE - Notice will be logged`
- `SWITCH: DEBUG - Debug infos will be logged`
- `SWITCH: ABOUT - About infos will be logged`
- `SWITCH: VERSION - Version infos will be logged`
- `SWITCH: INPUT - Input data will be logged (dangerous for passwords)`
- `SWITCH: PARAMETER - Parsed parameter will be logged`
- `SWITCH: STATISTIC - Statistic will be logged`
- `SWITCH: SUMMARY - Summary will be logged`
- `SWITCH: INFO - Requested information will be logged`
- `SWITCH: LICID - License ID will be logged`
- `SWITCH: LICENSE - License summary will be logged`
- `SWITCH: PROGRESS - Progress bar will be logged`
- `SWITCH: USAGE - Usage reporting will be logged`
- `SWITCH: CONDBG - Debug information for remote connections`
- `SWITCH: ALL - All messages are logged`
- `SWITCH: DEFAULT - Relevant messages are logged`
- `SWITCH: MINIMAL - Important messages are logged`
- `SWITCH: ERRONLY - Only errors and error trace are logged`

3.1.8. **OVERLAY** *DESTINATION*

SYNOPSIS

```
HELP:   Destination for log messages
PATH:   RUN.LOG
TYPE:   OVERLAY
SYNTAX: DESTINATION.{STREAM()/FILE()/SYSTEM()}
```

DESCRIPTION

The log destination parameter defines where the log messages will be written.

Each log message starts with the current time and date.

All the possible destinations allow to specify a custom identification string which will be printed after time and date of each message. This is done with the IDENT argument.

If no IDENT specified the default will be *owner.program.command*

If no log destination defined stream will be used as default.

3.1.9. OBJECT *STREAM*

SYNOPSIS

```
HELP:   Log to a stream
PATH:   RUN.LOG.DESTINATION
TYPE:   OBJECT
SYNTAX: STREAM(IDENT='str',FORMAT=STANDARD/DIALOG/MINIMAL,STDOUT)
```

DESCRIPTION

The STREAM object allows the log messages to be written to the standard output or the standard error output. Standard error output is used if STDOUT is not specified.

ARGUMENTS

- `STRING: IDENT='str' - Ident of component for logging [owner]`
- `NUMBER: FORMAT=STANDARD/DIALOG/MINIMAL - Format of log message [STANDARD]`
  - `STANDARD - Standard logging format (timestamp ident *level* message)`
  - `DIALOG - For dialog processing (ident *level* message)`
  - `MINIMAL - For output processing (message)`
- `SWITCH: STDOUT - Log messages to stdout [STDERR])`

3.1.10. OBJECT *FILE*

SYNOPSIS

```
HELP:   Log to a flat file
PATH:   RUN.LOG.DESTINATION
TYPE:   OBJECT
SYNTAX: FILE(IDENT='str',FORMAT=STANDARD/DIALOG/MINIMAL,NAME='str')
```

DESCRIPTION

The flat file option allows the log messages to be written to a normal file. The name of the file must be specified for this destination. The log messages will be appended to this file if it already exists.

For the log file, all rules and replacements are valid which are described under "FILENAME HANDLING" above. For example:

```
   NAME='~.MYLOG(LOG1)'                    ; PDS on z/OS
   NAME='<SYSUID>.MYLOG02'                 ; PS dataset on z/OS
   NAME='~/mylog3.txt'                     ; Unix path name
   NAME='<HOME>\logs\mylog4.txt'           ; Windows path name
```

ATTENTION: Parallel process cannot share the same log file

ARGUMENTS

- `STRING: IDENT='str' - Ident of component for logging [owner]`
- `NUMBER: FORMAT=STANDARD/DIALOG/MINIMAL - Format of log message [STANDARD]`
  - `STANDARD - Standard logging format (timestamp ident *level* message)`
  - `DIALOG - For dialog processing (ident *level* message)`
  - `MINIMAL - For output processing (message)`
- `STRING: NAME='str' - Filename to append log messages [''==stderr])`

3.1.11. OBJECT *SYSTEM*

SYNOPSIS

```
HELP:   Use system logging facility
PATH:   RUN.LOG.DESTINATION
TYPE:   OBJECT
SYNTAX: SYSTEM(IDENT='str')
```

DESCRIPTION

The SYSTEM destination allows the log messages to be written to the logging system of the operating

system.

On Unix and z/OS® systems the syslog() call is used. On Windows® the event logging facility is used.

```
NOTE: On Windows(R) this requires the registry key entry:
      HKEY_LOCAL_MACHINE
        SYSTEM
          CurrentControlSet
            Services
              EventLog
                Application
                  FLM
```

This registry key will be created by using the DLL flevent.dll

ARGUMENTS

- `STRING: IDENT='str' - Ident of component for logging [owner]`

Chapter 4. Available built-in functions

Numerous built-in functions are available to obtain extensive help about syntax, errors, commands and functions of a program with help messages, manual pages and documentation.

Other built-in functions can be used for owner, property, environment and trace management. All built-in functions have the purpose to simplify handling of powerful and sometimes complex commands.

4.1. FUNCTION *SYNTAX*

SYNOPSIS

```
HELP:   Provides the syntax for each command
PATH:   limes.flssrv
TYPE:   BUILT-IN FUNCTION
SYNTAX: :> flssrv SYNTAX [command[.path] [DEPTH1 | ... | DEPTH9 | ALL]]
```

DESCRIPTION

The syntax function shows how the arguments can be set on the command line. Depending on depth the syntax is printed in one line or in structured form. A structured print out means that you have a question or exclamation mark before the argument.

- question mark - optional parameter
- exclamation mark - required parameter

The **path** is a dotted keyword list in accordance with the syntax of the corresponding command.

```
command.para1.para2.para3
```

OPTIONS

```
DEPTH1 / DEPTH2 / ... / DEPTH9
```

Specifies the depth of the output. Default is DEPTH1. If the depth is greater than 1 the structured print out is used.

```
ALL
```

Shows complete syntax for the specified path, not just DEPTH1. Structured print out is used.

EXAMPLES

```
:> FLSSRV SYNTAX
:> FLSSRV SYNTAX command
:> FLSSRV SYNTAX command ALL
:> FLSSRV SYNTAX command.para
:> FLSSRV SYNTAX command.para ALL
```

4.2. FUNCTION *HELP*

SYNOPSIS

```
HELP:   Provides quick help for arguments
PATH:   limes.flssrv
TYPE:   BUILT-IN FUNCTION
SYNTAX: :> flssrv HELP [command[.path] [DEPTH1 | ... | DEPTH9 | ALL]] [MAN]
```

DESCRIPTION

The help function can provide extensive information about each parameter which is adjustable on the command line. With the help functionality you get a structured list of arguments:

- keyword - the keyword for the argument
- (TYPE: ...) - the type of this argument
- help message - the short help text
- (PROPERTY: [...]) - the current hard coded property value (optional)

The keyword can be abbreviated, but it must be unique for this list. The required letters are upper case if case sensitive mode is used. With the case sensitive mode an additional line is required to mark the appropriate letter.

The **path** is a dotted keyword list in accordance with the syntax of the corresponding command.

```
command.para1.para2.para3
```

Help can also be used to show the corresponding detailed description of the manual pages with keyword **MAN**.

OPTIONS

```
DEPTH1 / DEPTH2 / ... / DEPTH9
```

Specifies the depth of the output. Default is DEPTH1.

```
ALL
```

Shows complete help below specified path, not just DEPTH1 and includes all aliases defined for a certain argument.

```
MAN
```

Shows additionally the detailed description for this program, command or parameter and the next level of arguments including the aliases.

EXAMPLES

```
:> FLSSRV HELP
:> FLSSRV HELP MAN
:> FLSSRV HELP command
:> FLSSRV HELP command MAN
:> FLSSRV HELP command ALL
:> FLSSRV HELP command.para
:> FLSSRV HELP command.para MAN
:> FLSSRV HELP command.para ALL
```

4.3. FUNCTION *MANPAGE*

SYNOPSIS

```
HELP:   Provides manual pages (detailed help)
PATH:   limes.flssrv
TYPE:   BUILT-IN FUNCTION
SYNTAX: :> flssrv MANPAGE [function | command[.path][=filename]] | [filename]
```

DESCRIPTION

Shows the manual pages of the program, all commands and built-in functions and if available for each argument up to selections/constant definitions.

This function prints the corresponding section from the user manual on the screen (doctype: **book**). If a filename is given then the doctype **manpage** of ASCIIDOC is written to the file.

Do not use space characters between the key, assignment character (=) and filename.

The files in ASCIIDOC format can be used to generate manpages with the tool ASCIIDOC.

EXAMPLES

```
:> FLSSRV MANPAGE
:> FLSSRV MANPAGE filename
:> FLSSRV MANPAGE command
:> FLSSRV MANPAGE command=filename
:> FLSSRV MANPAGE function
:> FLSSRV MANPAGE function=filename
:> FLSSRV MANPAGE command.para
:> FLSSRV MANPAGE command.para=filename
```

4.4. FUNCTION *GENDOCU*

SYNOPSIS

```
HELP:   Generates auxiliary documentation
PATH:   limes.flssrv
TYPE:   BUILT-IN FUNCTION
SYNTAX: :> flssrv GENDOCU [command[.path]=]filename [NONBR]
```

DESCRIPTION

The complete user manual is generated when only the filename is given. If a command, or command path, is specified the corresponding section of the user manual is written to a file. The generated files are in ASCIIDOC format, to be readable as normal text document on all platforms and easily convertible into HTML, PDF or other formats with ASCIIDOC.

The numbering will vary in places where the complete manual uses a bullet list for arguments and constants, when no detailed description is available and some of these arguments are required for the path.

The numbering can be disabled by setting the optional keyword NONBR. This is useful if numbering is generated by a post processing utility (for example *pdflatex*).

Do not use space characters between the key, assignment character (=) and filename.

EXAMPLES

```
:> FLSSRV GENDOCU flssrv.manual.txt NONBR
:> FLSSRV GENDOCU command=command.manual.txt
:> FLSSRV GENDOCU command.argument=argument.manual.txt
```

4.5. FUNCTION *GENPROP*

SYNOPSIS

```
HELP:   Generates a property file
PATH:   limes.flssrv
TYPE:   BUILT-IN FUNCTION
SYNTAX: :> flssrv GENPROP [command=]filename
```

DESCRIPTION

A property file is generated for all commands if only a filename is provided. If a command is specified, a property file for this specific command is created. Property files can be used to define default settings for commands. For each parameter, a property entry is written to a text file. The text file contains comments to help in editing the properties. A property file can be activated with the built-in function SETPROP.

Property files will be activated by their owner. A property file overrides hard coded properties and properties defined over environment variables. A specific property file overrides the general property file. At time of generation all current default settings are written to the property file.

We recommend to work only with command specific property files, because the built-in function CHGPROP can be used to generate, update and activate a property file for a certain command.

Do not use space characters between the key, assignment character (=) and filename.

To optimize the program performance, one can remove all unnecessary properties.

See PROPERTIES for the current property file content.

EXAMPLES

```
:> FLSSRV GENPROP owner.general.properties
:> FLSSRV GENPROP command=owner.command.properties
```

4.6. FUNCTION *SETPROP*

SYNOPSIS

```
HELP:   Activate a property file
PATH:   limes.flssrv
TYPE:   BUILT-IN FUNCTION
SYNTAX: :> flssrv SETPROP [command=]filename
```

DESCRIPTION

The file name is set as current global property file when no command is specified. The file name is set

as current local property file when a command is given. The property file can be generated with GENPROP and will be activated for the current owner.

After generation you can activate the property file, but there will be no effect until you change the settings inside the property file, because the current default settings are written.

The file name can contain replacement/mapping rules (<HOME>/<USER>).

Do not use space characters between the key, assignment character (=) and filename.

We recommend to work only with command specific property files, because the built-in function CHGPROP can be used to generate, update and activate a property file for a certain command.

See PROPERTIES for the current property file content.

EXAMPLES

```
:> FLSSRV SETPROP general.properties
:> FLSSRV SETPROP command=command.properties
```

4.7. FUNCTION *CHGPROP*

SYNOPSIS

```
HELP:   Change a property value in the currently active property file
PATH:   limes.flssrv
TYPE:   BUILT-IN FUNCTION
SYNTAX: :> flssrv CHGPROP command [path[=value]]*
```

DESCRIPTION

The built-in function CHGPROP updates property values in the currently active (see SETPROP) property file for a dedicated command. If no property file is activated, a property file is generated (see GENPROP), updated with the provided property list and activated (see SETPROP).

The first argument of CHGPROP is the command name. All other arguments are key-value pairs of properties that should be updated. The key is internally prefixed with the root (owner.program.command.). The values are enclosed in double quotation marks. Multiple properties of a command can be updated at once by separating them with a space character (please do not use spaces between key, assignment character (=) and the value). This set of properties is parsed by the CLP. If no property is provided, the property file for the command is either parsed or generated (if it does not exist) and then activated without updating properties. This behavior can be used to generate and activate command specific property files for the different commands.

The built-in function reads and parses the currently active property file. Then it parses the provided property list and writes the updated property file back to disk. The currently active property file could be a global, not command specific property file. In such a case, a new command specific property file is generated and activated. Properties from the global property file are copied into the newly generated one.

If no property file is defined (see SETPROP) that corresponds to the owner, program and command, a

property file is generated. The filename formats below are used to read or write property files from/to the home directory:

On non-mainframe systems (WINDOWS, UNIX, MAC): ".owner.program.command.properties"

On mainframe systems (ZOS, VSE, BS200, ...): "<SYSUID>.OWNER.PROGRAM.COMMAND.PROPS"

If the default filenames and/or path are not sufficient, you can change the filename of each property file by setting environment variables using the following naming convention:

```
OWNER_PROGRAM_COMMAND_PROPERTY_FILENAME
```

If no home directory is defined, then the current working directory is used as default directory to save the property files. On mainframes SYSUID is used for the first level qualifier, to represent the "home directory".

Property files are managed per owner: This means that updates are only done for the current owner.

To delete a property you can simply specify the path without the root and with out a sign.

See PROPERTIES for the current property file content.

EXAMPLES

```
:> FLSSRV CHGPROP command overlay.object.argument=value
:> FLSSRV CHGPROP command overlay.object.argument
:> FLSSRV CHGPROP command object.argument=value1 argument=value2
```

4.8. FUNCTION *DELPROP*

SYNOPSIS

```
HELP:   Remove a property file from configuration
PATH:   limes.flssrv
TYPE:   BUILT-IN FUNCTION
SYNTAX: :> flssrv DELPROP [command]
```

DESCRIPTION

Without a command the current global property file is deleted from the configuration data. A command specific property file can be deleted from the configuration data when the command name is given.

Property files are managed per owner, this means that delete is only done for the current owner.

We recommend to work only with command specific property files.

See PROPERTIES for the current property file content.

EXAMPLES

```
:> FLSSRV DELPROP
:> FLSSRV DELPROP command
```

4.9. FUNCTION *GETPROP*

SYNOPSIS

```
HELP:   Show current properties
PATH:   limes.flssrv
TYPE:   BUILT-IN FUNCTION
SYNTAX: :> flssrv GETPROP [command[.path] [DEPTH1 | ... | DEPTH9 | DEPALL | DEFALL]]
```

DESCRIPTION

All or certain properties can be shown for a dedicated path. The **path** is a dotted keyword list according to the syntax of the corresponding command.

```
command.para1.para2.para3
```

A help message for a certain parameter will also show the assigned property value.

See PROPERTIES for the current property file content.

OPTIONS

```
DEPTH1 / DEPTH2 / ... / DEPTH9
```

Specifies the depth of the output. Default is DEPTH1.

```
DEPALL
```

Shows all properties to the specified path, not just DEPTH1.

```
DEFALL
```

Shows all defined properties to the specified path.

EXAMPLES

```
:> FLSSRV GETPROP
:> FLSSRV GETPROP command
:> FLSSRV GETPROP command DEFALL
:> FLSSRV GETPROP command.para
:> FLSSRV GETPROP command.para DEPALL
```

4.10. FUNCTION *SETOWNER*

SYNOPSIS

```
HELP:   Defines the current owner
PATH:   limes.flssrv
TYPE:   BUILT-IN FUNCTION
SYNTAX: :> flssrv SETOWNER name
```

DESCRIPTION

The current owner id (prefix or separation) can be changed. The owner concept is very powerful to separate different property settings. You can run the same command with different properties simply by switching the owner. The owner is a general solution to separate all settings and can be used for environment, client and other purposes.

If not already defined, the owner is written to the environment table under the key word OWNERID. This environment variable can for example used as replacement in file names (~/<OWNERID>.dat) or to refer a signature key.

EXAMPLES

```
:> FLSSRV SETOWNER com.company
```

4.11. FUNCTION *GETOWNER*

SYNOPSIS

```
HELP:   Show current owner setting
PATH:   limes.flssrv
TYPE:   BUILT-IN FUNCTION
SYNTAX: :> flssrv GETOWNER
```

DESCRIPTION

The current owner id (prefix for separation) will be shown. The owner concept allows to manage different configurations as a logical separation of what is done with this id.

EXAMPLES

```
:> FLSSRV GETOWNER
```

4.12. FUNCTION *SETENV*

SYNOPSIS

```
HELP:   Set an environment variable
PATH:   limes.flssrv
TYPE:   BUILT-IN FUNCTION
SYNTAX: :> flssrv SETENV variable=value
```

DESCRIPTION

This function allows to set environment variables within the configuration file for a certain owner. All defined environment variables are placed in the current environment table of the process before a command is executed.

Do not use space characters between the key, assignment character (=) and value.

Attention: The environment variables will be temporary overwritten

EXAMPLES

```
:> FLSSRV SETENV LANG=DE_DE.IBM-1141
:> FLSSRV SETENV LANG=DE_DE.UTF-8
:> FLSSRV SETENV ENVID=T
:> FLSSRV SETENV ENVID=P
```

4.13. FUNCTION *GETENV*

SYNOPSIS

```
HELP:   Show the environment variables
PATH:   limes.flssrv
TYPE:   BUILT-IN FUNCTION
SYNTAX: :> flssrv GETENV
```

DESCRIPTION

This function lists all environment variables which are set within the configuration file for a certain owner. All shown environment variables are placed in the environment table of the process.

Attention: The environment variables will be temporary overwritten

EXAMPLES

```
   :> FLSSRV GETENV
```

4.14. FUNCTION *DELENV*

SYNOPSIS

```
HELP:   Delete an environment variable
PATH:   limes.flssrv
TYPE:   BUILT-IN FUNCTION
SYNTAX: :> flssrv DELENV variable
```

DESCRIPTION

This function will delete environment variables from the configuration file for a certain owner.

EXAMPLES

```
   :> FLSSRV DELENV LANG
```

4.15. FUNCTION *TRACE*

SYNOPSIS

```
HELP:   Manage trace capabilities
PATH:   limes.flssrv
TYPE:   BUILT-IN FUNCTION
SYNTAX: :> flssrv TRACE ON | OFF | FILE=filename
```

DESCRIPTION

Tracing for command line parser and commands can be enabled or disabled. A trace is only required to diagnose errors. In such a case a trace file must be defined first, then the trace can be activated, the erroneous command executed and the trace file sent to the support team.

If no trace file is defined, stdout/stderr is used as default. To prevent tracing on the screen, please define a trace file first before tracing is switched on.

Do not use space characters between the key, assignment character (=) and filename.

EXAMPLES

```
   :> FLSSRV TRACE FILE=filename
   :> FLSSRV TRACE ON
   :> FLSSRV TRACE OFF
```

4.16. FUNCTION *CONFIG*

SYNOPSIS

```
HELP:   Shows or clear all the current configuration settings
PATH:   limes.flssrv
TYPE:   BUILT-IN FUNCTION
SYNTAX: :> flssrv CONFIG [CLEAR]
```

DESCRIPTION

The actual configuration list with owner id, property file names trace setting, environment variables and more is shown. With the option CLEAR the configuration data including the corresponding file will be deleted.

The configuration file name can be defined by the environment variable:

```
FLSSRV_CONFIG_FILE=config.file.name
```

This can be used to work with a global configuration file. If this location is not defined, the configuration file is stored in the home directory with the name:

```
.flssrv.config
```

If no home directory is defined for the current user (this case is very improbable), the working directory is used.

To read the configuration file when the environment variable above is not defined, CLE first looks in the working and then in the home directory. This makes it possible to prefer a property file, by copying it to the current working directory.

On mainframes the data set name *SYSUID.FLSSRV.CONFIG* is used as default. SYSUID as high level qualifier is a kind of replacement for the current home directory in other operating systems in this way.

EXAMPLES

```
:> FLSSRV CONFIG
:> FLSSRV CONFIG CLEAR
```

4.17. FUNCTION *GRAMMAR*

SYNOPSIS

```
HELP:   Shows the grammar for commands and properties
PATH:   limes.flssrv
TYPE:   BUILT-IN FUNCTION
SYNTAX: :> flssrv GRAMMAR
```

DESCRIPTION

The grammar used for interpreting the command line and property files is shown. This is helpful for understanding how the commands and property files are interpreted by the command line processor.

See GRAMMAR for an explanation and the output of this command.

EXAMPLES

```
:> FLSSRV GRAMMAR
```

4.18. FUNCTION *LEXEM*

SYNOPSIS

```
HELP:   Shows the regular expressions accepted in a command
PATH:   limes.flssrv
TYPE:   BUILT-IN FUNCTION
SYNTAX: :> flssrv LEXEM
```

DESCRIPTION

The regular expression used for the tokens of the command syntax will be shown. This could be helpful in understanding how the commands and property files are interpreted by the command line processor. Moreover it will be shown how to define a number, a string, a keyword and other values inside a command.

See LEXEM for an explanation and the output of this command.

EXAMPLES

```
:> FLSSRV LEXEM
```

4.19. FUNCTION *LICENSE*

SYNOPSIS

```
HELP:   List license information for the program
PATH:   limes.flssrv
TYPE:   BUILT-IN FUNCTION
SYNTAX: :> flssrv LICENSE
```

DESCRIPTION

The license string of the software product is shown. This string is provided to the command line executer (CLE) by the calling program and should reflect the current license text for this program.

EXAMPLES

```
:> FLSSRV LICENSE
```

4.20. FUNCTION *VERSION*

SYNOPSIS

```
HELP:   List version information for the program
PATH:   limes.flssrv
TYPE:   BUILT-IN FUNCTION
SYNTAX: :> flssrv VERSION
```

DESCRIPTION

The version of the software product is shown. This string is provided to the command line executer (CLE) by the calling program and should reflect the current version information for this program.

See VERSION for the output of this command.

EXAMPLES

```
   :> FLSSRV VERSION
```

4.21. FUNCTION *ABOUT*

SYNOPSIS

```
HELP:   Show information about the program
PATH:   limes.flssrv
TYPE:   BUILT-IN FUNCTION
SYNTAX: :> flssrv ABOUT
```

DESCRIPTION

Shows information about the software product like version, copyright, external libraries, license and other attributes. This string is provided to the command line executer (CLE) by the calling program and should reflect the current about information for this program (flssrv).

See ABOUT for the output of this command.

EXAMPLES

```
   :> FLSSRV ABOUT
```

4.22. FUNCTION *ERRORS*

SYNOPSIS

```
HELP:   Show information about return and reason codes of the program
PATH:   limes.flssrv
TYPE:   BUILT-IN FUNCTION
SYNTAX: :> flssrv ERRORS
```

DESCRIPTION

This built-in function prints explanations for the return/condition/exit codes of the executable and, if available, the reason codes of executed commands.

The command line executer (CLE) initialized the CLP structure. Calls the command line processor (CLP). Calls the mapping function, which converts the CLP structure in a corresponding command structure. Then it runs the command and as last step it calls the finish function with the command structure, to close all files and free all the allocated memory. If one of these steps fails, then a corresponding return (condition/exit) code is given back from the executable. Only the command execution can result in a error (return code 8) or in a warning (return code 4). The command it self must return a reason code for such a warning or error. This means that the return (condition/exit) code can be used to control the batch processing. The reason code gives more information about the error or warning.

See RETURN CODES and REASON CODES for an explanation and the output of this command.

EXAMPLES

```
:> FLSSRV ERRORS
```

Appendix A: LEXEM

The regular expressions used by the command line compiler are shown below. This content is printed on the screen with the built-in function *LEXEM*.

To describe the regular expression the following character classes are used:

- :print: - all printable characters
- :space: - all blanks, tabs and new lines
- :cntr: - all control characters
- :alpha: - all letters ([[a-z]|[A-Z]])
- :alnum: - all letters and digits [0-9]
- :digit: - all decimal numbers [0-9]
- :xdigit: - all hexadecimal numbers [[0-9]|[a-f||[A-F]]

All comments and separators are ignored by the scanner and not passed with the token to the parser. This means that the separators (SEP) are used in the grammar but will never be part of or own tokens.

If a binary entry for a string is possible and no classification is done it is taken as a binary string in local character set representation (c) without null termination. This default can be changed. In this case the help message for this argument should be described the default interpretation of the binary string. For binary strings it is still possible to enter a null-terminated string, but then the prefix **s** must be used. If no binary entry is feasible, only a null-terminated string in local character representation (s) can be built and taken by default.

```
Lexemes (regular expressions) for argument list or parameter file
--| COMMENT   '#' [:print:]* '#'                              (will be ignored)
--| LCOMMENT  ';' [:print:]* 'nl'                             (will be ignored)
--| SEPARATOR [:space: | :cntr: | ',']*                  (abbreviated with SEP)
--| OPERATOR1 '=' | '.' | '(' | ')' | '[' | ']'  (SGN, DOT, RBO, RBC, SBO, SBC)
--| OPERATOR2 '+' | '-' | '*' | '/'                        (ADD, SUB, MUL, DIV)
--| KEYWORD   ['-'['-']][:alpha:]+[:alnum: | '_']*          (always predefined)
--| NUMBER    ([+|-]  [ :digit:]+)  |                       (decimal (default))
--| num       ([+|-]0b[ :digit:]+)  |                                 (binary)
--| num       ([+|-]0o[ :digit:]+)  |                                  (octal)
--| num       ([+|-]0d[ :digit:]+)  |                                (decimal)
--| num       ([+|-]0x[ :xdigit:]+) |                            (hexadecimal)
--| num       ([+|-]0t(yyyy/mm/tt.hh:mm:ss)) |  (relativ (+|-) or absolut time)
--| FLOAT     ([+|-]  [ :digit:]+.[:digit:]+e|E[:digit:]+) | (decimal(default))
--| flt       ([+|-]0d[ :digit:]+.[:digit:]+e|E[:digit:]+)           (decimal)
--| STRING         ''' [:print:]* ''' |          (default (if binary c else s))
--| str       [s|S]''' [:print:]* ''' |                (null-terminated string)
--| str       [c|C]''' [:print:]* ''' |  (binary string in local character set)
--| str       [a|A]''' [:print:]* ''' |                (binary string in ASCII)
--| str       [e|E]''' [:print:]* ''' |               (binary string in EBCDIC)
--| str       [x|X]''' [:print:]* ''' |         (binary string in hex notation)
--| str       [f|F]''' [:print:]* ''' | (read string from file (for passwords))
--|           Strings can contain two '' to represent one '
```

```
--|           Strings can also be enclosed in " or ` instead of '
--|           Strings can directly start behind a '=' without enclosing '/"
--|              In this case the string ends at the next separator or operator
--|              and keywords are preferred. To use keywords, separators or
--|              operators in strings, enclosing quotes are required.
--|
--| The predefined constant keyword below can be used in a value expressions
--| NOW       NUMBER - current time in seconds since 1970 (+0t0000)
--| MINUTE    NUMBER - minute in seconds (60)
--| HOUR      NUMBER - hour in seconds   (60*60)
--| DAY       NUMBER - day in seconds    (24*60*60)
--| YEAR      NUMBER - year in seconds   (365*24*60*60)
--| KiB       NUMBER - kilobyte          (1024)
--| MiB       NUMBER - megabyte          (1024*1024)
--| GiB       NUMBER - gigabyte          (1024*1024*1024)
--| TiB       NUMBER - terrabyte         (1024*1024*1024*1024)
--| RNDn      NUMBER - simple random number with n * 8 bit in length (1,2,4,8)
--| PI        FLOAT  - PI (3.14159265359)
--| LCSTAMP   STRING - current local stamp in format:           YYYYMMDD.HHMMSS
--| LCDATE    STRING - current local date in format:            YYYYMMDD
--| LCYEAR    STRING - current local year in format:            YYYY
--| LCYEAR2   STRING - current local year in format:            YY
--| LCMONTH   STRING - current local month in format:           MM
--| LCDAY     STRING - current local day in format:             DD
--| LCTIME    STRING - current local time in format:            HHMMSS
--| LCHOUR    STRING - current local hour in format:            HH
--| LCMINUTE  STRING - current local minute in format:          MM
--| LCSECOND  STRING - current local second in format:          SS
--| GMSTAMP   STRING - current Greenwich mean stamp in format:  YYYYMMDD.HHMMSS
--| GMDATE    STRING - current Greenwich mean date in format:   YYYYMMDD
--| GMYEAR    STRING - current Greenwich mean year in format:   YYYY
--| GMYEAR2   STRING - current Greenwich mean year in format:   YY
--| GMMONTH   STRING - current Greenwich mean month in format:  MM
--| GMDAY     STRING - current Greenwich mean day in format:    DD
--| GMTIME    STRING - current Greenwich mean time in format:   HHMMSS
--| GMHOUR    STRING - current Greenwich mean hour in format:   HH
--| GMMINUTE  STRING - current Greenwich mean minute in format: MM
--| GMSECOND  STRING - current Greenwich mean second in format: SS
--| GMSECOND  STRING - current Greenwich mean second in format: SS
--| SnRND10   STRING - decimal random number of length n (1 to 8)
--| SnRND16   STRING - hexadecimal random number of length n (1 to 8)
--|
--| SUPPLEMENT     '"' [:print:]* '"' |   (null-terminated string (properties))
--|           Supplements can contain two "" to represent one "
--|           Supplements can also be enclosed in ' or ` instead of "
--|           Supplements can also be enclosed in ' or ` instead of "
--| ENVIRONMENT VARIABLES '<'varnam'>' will replaced by the corresponding value
--| Escape sequences for critical punctuation characters on EBCDIC systems
--|    '!' = '&EXC;'   - Exclamation mark
--|    '$' = '&DLR;'   - Dollar sign
```

```
--|    '#' = '&HSH;'   - Hashtag (number sign)
--|    '@' = '&ATS;'   - At sign
--|    '[' = '&SBO;'   - Square bracket open
--|    '\' = '&BSL;'   - Backslash
--|    ']' = '&SBC;'   - Square bracket close
--|    '^' = '&CRT;'   - Caret (circumflex)
--|    '`' = '&GRV;'   - Grave accent
--|    '{' = '&CBO;'   - Curly bracket open
--|    '|' = '&VBR;'   - Vertical bar
--|    '}' = '&CBC;'   - Curly bracket close
--|    '~' = '&TLD;'   - Tilde
--| Define CCSIDs for certain areas in CLP strings on EBCDIC systems (0-reset)
--|    '&' [:digit:]+ ';  (..."&1047;get.file='&0;%s&1047;'",f)
--| Escape sequences for hexadecimal byte values
--|    '&' ['X''x'] :xdigit: :xdigit: ';' ("&xF5;")
```

Appendix B: GRAMMAR

Below you can see the grammar used by the command line compiler. This content is printed on the screen if the built-in function *GRAMMAR* is used.

There are two context free grammars, one is used for the command line the other for the parameter file. The parameter file contains mainly the same argument list as the command line. The property file defines the default settings. For this a different grammar is applied.

Properties can be set for each argument, i.e. for arguments not used or arguments in overlays. Based on that, properties are set by using the path (dotted keyword list).

A property is always a *SUPPLEMENT* for the argument list on the command line or parameter file. Thus all defined properties have the same effect as arguments on the command line. A property sets this argument to the corresponding value if it is not defined on the command line.

```
Grammar for argument list, parameter file or property file
--| Command Line Parser
--| command        -> ['('] parameter_list [')']       (main=object)
--|                |  ['.'] parameter                   (main=overlay)
--| parameter_list -> parameter SEP parameter_list
--|                |  EMPTY
--| parameter      -> switch | assignment | object | overlay | array
--| switch         -> KEYWORD
--| assignment     -> KEYWORD '=' value
--|                |  KEYWORD '=' KEYWORD # SELECTION #
--| object         -> KEYWORD ['('] parameter_list [')']
--|                |  KEYWORD '=' STRING # parameter file #
--| overlay        -> KEYWORD ['.'] parameter
--|                |  KEYWORD '=' STRING # parameter file #
--| array          -> KEYWORD '[' value_list   ']'
--|                |  KEYWORD '[' object_list  ']'
--|                |  KEYWORD '[' overlay_list ']'
--|                |  KEYWORD '[=' STRING ']' # parameter file #
--| value_list     -> value SEP value_list
--|                |  EMPTY
--| object_list    -> object SEP object_list
--|                |  EMPTY
--| overlay_list   -> overlay SEP overlay_list
--|                |  EMPTY
--| A list of objects requires parenthesis to enclose the arguments
--|
--| value          -> term '+' value
--|                |  term '-' value
--|                |  term
--| term           -> factor '*' term
--|                |  factor '/' term
--|                |  factor
--| factor         -> NUMBER | FLOAT | STRING
--|                |  selection | variable | constant
```

```
--|                |  '(' value ')'
--| selection      -> KEYWORD # value from a selection table        #
--| variable       -> KEYWORD # value from a previous assignment    #
--|                |  KEYWORD '[' value ']' # with index for arrays #
--| constant       -> KEYWORD # see predefined constants at lexem   #
--| For strings only the operator '+' is implemented as concatenation
--| Strings without an operator in between are also concatenated
--| A number followed by a constant is a multiplication (4KiB=4*1024)
--|
--| Property File Parser
--| properties     -> property_list
--| property_list  -> property SEP property_list
--|                |  EMPTY
--| property       -> keyword_list '=' SUPPLEMENT
--| keyword_list   -> KEYWORD '.' keyword_list
--|                |  KEYWORD
--| SUPPLEMENT is a string in double quotation marks ("property")
```

Appendix C: PROPERTIES

All properties can be written to a global property file with the built-in function:

```
'GENPROP filename'.
```

A property file for a certain command can be generated with:

```
'GENPROP command=filename'
```

The first comment shows the valid root path for the property file. A property file can contain properties for more than one root path. The property file parser will only interpret properties where the root matches, i.e. if the property file is assigned to a certain owner and possibly to a certain command, then only properties for this owner, the program itself and possibly the command will be accepted, all other properties are ignored.

The second long comment gives some help for handling a property file. Thereafter the generated properties for this root are listed for all supported commands. The format is:

```
path = SUPPLEMENT # Type: tttttt Help: hhh...hhh #
```

The path denotes the unique dotted keyword list for one argument. The *SUPPLEMENT* is a string normally in double quotation marks containing the value list for this argument in accordance with the syntax of the command line parser (strings then normally in single quotation marks, a opposite usage of the quotation marks is possible but not recommended). This means for example that a string that requires apostrophes must be included in apostrophes (`"'...'"`). The supplement string is returned by the corresponding command line parser if no value for this argument is entered on the command line or is included in the parameter file.

Behind the property definition the type and help message are printed as comment to help in editing the generated property file. The property values can be related to the current default settings at time of generation, what means that if hard coded default values or another property file is already in use, then these values are taken over into the new property file.

The property file can contain parameter which are not available on the command line. Such parameter are only documented in the property file. On the other side not all command line parameter are available as property.

If a description for a property argument is available which is not a part of the user manual this description is written as comment in front of the corresponding property definition.

Below you can see the documentation of all parameter which are only available as property followed by all defined default properties in form of a sample property file for all available commands.

C.1. REMAINING DOCUMENTATION

Below you can find the documentation for all parameters which are only available over the property

definitions. These parameters are not a part of the description for each command above. This list is generated and can be empty. In this case all parameters are available over the command line. If available, the manual page otherwise the help message is printed.

C.2. PREDEFINED DEFAULTS

Below you can find a sample property file which contains all predefined default values for all supported commands when this documentation was created.

```
# Property file for: limes.flssrv #
#------------------------------------------------------------------ #
#                                                                   #
# The property file can be used to overwrite the default values     #
# used if the argument is not defined on the command line.          #
#                                                                   #
#------------------------------------------------------------------ #
#                                                                   #
# The values is assigned between "" conforming to the syntax.       #
#   Example NUMBER array  : "4711, 1, 2, 3"                         #
#   Example STRING array  : "'str1' x'303000''str2',a'abc'"         #
#   Example FLOAT array   : "123.34, 1.58, PI" (PI as constant)     #
#                                                                   #
# The last example shows that one can use key words if a list       #
# of constants is defined for this argument.                        #
#                                                                   #
# For a SWITCH one can use the special key words ON/OFF to enable or #
# disable the SWITCH or define the value as a number                #
#   Examples for a SWITCH : "ON"/"OFF"/"4711"                       #
#                                                                   #
# For an OBJECT only the special key word INIT can be used to ensure #
# that this OBJECT will be initialized if it was not set on the     #
# command line. If INIT is not set, then the elements of the OBJECT #
# will be initialized only if you type at least obj() in the command #
# line. If you want an object to be initialized when you don't use it#
# on the command line then the special key word must be set.        #
#   Examples for an OBJECT: ""/"INIT"                               #
#                                                                   #
# For an OVERLAY it is the same logic as for objects but the keyword #
# of the element to initialize must be used if the overlay is not   #
# set in the command line.                                          #
#   Example OVERLAY array : "LINE TRIANGLE LINE RECTANGLE"          #
#   Example for a OVERLAY : "RECTANGLE"                             #
#                                                                   #
#------------------------------------------------------------------ #
```

Appendix D: RETURN CODES

The return codes of the program are also called completion codes and returned at program termination. This content is printed on the screen with the built-in function *ERRORS*.

- 0 - command line, command syntax, mapping, execution and finish of the command was successful
- 1 - command line, command syntax, mapping, execution and finish of the command was successful but a warning can be found in the log
- 2 - command line, command syntax, mapping, execution was successful but cleanup of the command failed (may not happened)
- 4 - command line, command syntax and mapping was successful but execution of the command returned with a warning
- 8 - command line, command syntax and mapping was successful but execution of the command returned with an error
- 12 - command line and command syntax was OK but mapping of the parameter for command execution failed
- 16 - command line was OK but command syntax was wrong (error from command line parser)
- 20 - command string was wrong (please use built-in function "SYNTAX" for more information)
- 24 - initialization of the parameter structure for the command failed (may not happened)
- 28 - configuration is wrong (determination of configuration data failed, configuration file might be damaged)
- 32 - table error (something within the predefined tables is wrong (internal error))
- 36 - system error (e.g. load of environment or open of file failed)
- 40 - access control or license error
- 44 - interface error (e.g. parameter pointer equals NULL (only at call of subprogram interface))
- 48 - memory allocation failed (e.g. dynamic string handling)
- 64 - fatal error (basic things are damaged)
- >64 - special condition codes from the command for job control

The command line is interpreted by CLE the command string by CLP. For example:

```
:> flcl conv "read.file='test.txt' write.flam(file='test.adc')"
   |    |     |-- the command string is compiled by CLP
   |    |     |   and might result in a syntax error
   |    |-- conv is part of the command line
   |-- flcl is part of the command line

:> flcl conv=para.txt
   |    |    |-- para.txt is part of the command line, but the content
   |    |        of para.txt might contain a CLP syntax error
   |    |-- conv is part of the command line
   |-- flcl is part of the command line
```

A command execution (run) will return 0 or 1 on success, 4 if a warning occurs (only some of the files

converted), 8 in case of an error or special condition codes greater then 64 can be returned.

If the command execution was successful (return code 0) but a relevant warnings was logged then the return code is 1 (anything was fine, but please have a look in the log (e.g. a key is close to expire)). A return code of 2 indicates that a call to the finish function failed. Such a fail of the finish function is not recognizable if the run function (command execution) returns with a warning or an error. On the other side, if the finish failed then an additional warning in the log cannot determined because the condition code is still 2. All completion codes bigger than 8 and smaller or equal to 64 indicate an error in front of the command execution.

A relevant warning is a warning written to the log by a used component. Warnings written by components which are removed from procession because the format does not fit or something else, are still in the log but not relevant for the completion code 1.

Appendix E: REASON CODES

Below you can find the list of reason codes returned after command execution. Each command returns a reason code for the corresponding warning (return code 4) or error (return code 8). This content is printed on the screen with the built-in function *ERRORS*.

- 1 - Fatal error (may not happen)
- 2 - System error (wrong size of data types)
- 3 - Handle error (not in right shape)
- 4 - Interface error (parameter incorrect)
- 5 - Activity not possible
- 6 - Memory allocation failed (not enough memory)
- 7 - Length error (need more data at read or more space at write)
- 8 - Control code wrong
- 9 - Version not supported
- 10 - Function not supported
- 11 - Mode not supported
- 12 - Suite not supported
- 13 - Confidentiality mode not supported
- 14 - Integrity mode not supported
- 15 - Verification mode not supported
- 16 - Wrong header checksum
- 17 - Wrong pot checksum
- 18 - Wrong data checksum
- 19 - Wrong checksum
- 20 - Wrong message authentication code (MAC)
- 21 - Wrong key test value
- 22 - Syntax error (file not formatted correctly)
- 23 - Count error (supplied or calculated amount wrong)
- 24 - End of list (handle points behind last or first element)
- 25 - Delete of file failed
- 26 - Type error (supplied or calculated type wrong)
- 27 - Rebuild required (flush not possible)
- 28 - No change (informational)
- 29 - Status error (handle not in right state)
- 30 - Orientation not supported
- 31 - Split method not supported
- 32 - Maximum delimiter amount achieved
- 33 - Error at compression
- 34 - Error at decompression

- 35 - Maximum header length achieved
- 36 - Maximum data length achieved
- 37 - Open of FLAMFILE failed
- 38 - Record format not supported
- 39 - Writing to file failed
- 40 - Reading from file failed
- 41 - Get position of file pointer failed
- 42 - Set position of file pointer failed
- 43 - End of file (informational)
- 44 - Rename of FLAMFILE failed
- 45 - Delete of FLAMFILE failed
- 46 - Close of FLAMFILE failed
- 47 - Seek in file does not work
- 48 - Set of buffer size for FIO failed
- 49 - FLAMFILE format not supported
- 50 - No FLAMFILE
- 51 - No entry found
- 52 - Construct is empty
- 53 - Connection error
- 54 - Unsupported FLAM-ID
- 55 - Remote procedure call failed
- 56 - Method not supported
- 57 - Open of original file failed
- 58 - Error from ZLIB (GZIP)
- 59 - Character set not supported
- 60 - Space too small
- 61 - Logging facility not available
- 62 - Error from character conversion module
- 63 - Open of hash output file failed
- 64 - Close of original file failed
- 65 - Error with FKME
- 66 - Error at character conversion
- 67 - Info not supported
- 68 - Band not supported
- 69 - Open of report file failed
- 70 - Size error (supplied or calculated size wrong)
- 71 - Invalid character encountered
- 72 - Incomplete character encountered
- 73 - Error with byte order mark (BOM)
- 74 - Format wrong or not supported

- 75 - Auto conversion not possible
- 76 - Error from BZIP2 library
- 77 - Environment variable are not or cannot be defined
- 78 - Dynamic load of a function failed
- 79 - Access control or license violation
- 80 - Compression limit not reached
- 81 - Error from LZMA/XZ library
- 82 - Error from XML parser (EXPAT)
- 83 - Whole first block is padding
- 84 - Command line parser failed
- 85 - Mapping failed
- 86 - Open of temporary file failed
- 87 - Wrong magic bytes
- 88 - Corrupted or no FLAMFILE
- 89 - Initialization error
- 90 - Parameter not formatted correctly
- 91 - Binary data detected
- 92 - From and to are equal
- 93 - Data is encrypted (key required)
- 94 - Key value wrong
- 95 - Reallocation required
- 96 - Invalid username
- 97 - Authentication failed
- 98 - Hash calculation failed
- 99 - Processing of one or more files failed
- 100 - Hash/checksum file is empty
- 101 - Read of hash/checksum file failed
- 102 - Name of hash/checksum file is not valid
- 103 - Hash/checksum file is not formatted correctly
- 104 - Open of code table failed
- 105 - Code table is not formatted correctly
- 106 - Attribute settings not supported
- 107 - Option wrong or not supported
- 108 - Invalid parameter combination
- 109 - Open of info file failed
- 110 - Protocol wrong or not supported
- 111 - Differences found
- 112 - Try again with an ASCII code page (LATIN1) for read
- 113 - Data incomplete
- 114 - Procedure not supported

- 115 - Key not found
- 116 - OpenPGP error
- 117 - Algorithm not supported
- 118 - Key is inactive
- 119 - Key is invalid
- 120 - Key is expired
- 121 - Positioning failed
- 122 - Sequence error, incorrect order
- 123 - Update not possible
- 124 - Regular expression error
- 125 - Value out of range
- 126 - Usage measurement not possible
- 127 - Dummy processing not possible
- 128 - Open of inverse file failed
- 129 - Internal error (may not happen)
- 130 - Micro Focus support failed
- 131 - Dynamic string library error
- 132 - Open of usage log failed
- 133 - Serialization failed
- 134 - Deserialization failed
- 135 - Program call (PC) failed
- 136 - Row specification wrong
- 137 - Record delimiter not supported
- 138 - End of table (informational)
- 139 - End of member (informational)
- 140 - Have more data (informational)
- 141 - Alignment error
- 142 - Not enough space available (internal buffer too small)
- 143 - Plausibility error
- 144 - Unexpected remaining rest (data don’t fit within processing)
- 145 - Length parameter too small (the resulting space is not sufficient)
- 146 - Virus found
- 147 - Connection was closed
- 148 - Error from anti virus scanner

Appendix F: VERSION

Here you can see the version of the program and its components. This content is printed on the screen if the built-in function *VERSION* is used.

```
00 FLIES-SEVER VERSION: 5.1.20-23975 BUILD: RELEASE May 10 2019 19:58:58
```

Appendix G: ABOUT

This is the about information for the program (flssrv). Its content is printed on the screen if the built-in function *ABOUT* is used.

```
00 Frankenstein Limes Integrated Extended Security Server (FLIES(R) Server)
01 FLIES-SEVER VERSION: 5.1.20-23975 BUILD: RELEASE May 10 2019 19:58:58
   Copyright (C) limes datentechnik (R) gmbh
   All rights reserved
The FLIES(R) server is a convenient method to access a remote FLAMARCHIVE.
On LINUX systems we use additionally the glibc as part of the operating system
   FLIES(R) server is a work that uses the library glibc in unmodified form based on
the LGPL license
   Copyright (C) 1991-2012 Free Software Foundation, Inc.
   More information to this license you can find under:
      http://www.fsf.org/
      http://www.gnu.org/licenses/lgpl-3.0.txt
```

INDEX

COLOPHON

limes datentechnik(R) gmbh
Louisenstrasse 101
D-61348 Bad Homburg v.d.H.
phone: +49(0)6172-5919-0
fax: +49(0)6172-5919-39
mail: info@flam.de
web: www.flam.de or www.limes.de
Amtsgericht: Bad Homburg vor der Hoehe HRB 3288 (gegr. 1985)
Geschaeftsfuehrer: Diplom-Mathematiker Heinz-Ulrich Wiebach
limes datentechnik(R): efficiency at the limit of possibility.