FAME_library

BLOODROCK/tRSi/F-Innovation

COLLABORATORS

	TITLE : FAME_library	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	BLOODROCK/tRSi/F-Innovation	February 12, 2023	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

FAME_library

1.1 FAME.library/FAMEAdd64

```
                NAME
      FAMEAdd64       -- perform a QuadWord (64-bit) addition

SYNOPSIS
      FAMEAdd64(SrcHi, SrcLo, Destination)
                 d0     d1        a0

      VOID FAMEAdd64(ULONG, ULONG, APTR)

FUNCTION
      Adds an unsigned 8-byte value (SrcLo/SrcHi) to a QuadWord at a
      specified destination address.

INPUTS
      SrcHi           - The upper LONG of the 64-bit value to add.
                        If you want to add a LONG to the destination
                        value (instead of a QuadWord), set SrcHi to NULL
                        and pass the LONG to be added in SrcLo.

      SrcLo           - The lower LONG of the 64-bit value to add.

      Destination     - A pointer to a memory address of 8 bytes
                        in size where SrcHi and SrcLo get added to.
                        These 8 bytes build one QuadWord. If the first
                        half of this area (the first LONG) is NULL,
                        the QuadWord value equals the lower LONG
                        value and will be inside a normal range from
                        0 to 4,294,967,295.
                        If the upper LONG is non-zero, the 64-bit value
                        (Destination) equals:
                        (UpperLong * 4,294,967,296) + LowerLong.

EXAMPLE
      Destination may be a structure, or a part of a structure:

              struct  MyData {
              ULONG   ULBytesHi;
              ULONG   ULBytesLo;
```

```
        };

    The function call may look like this:

    FAMEAdd64(NULL, BytesForThisUpload, &MyData)

NOTE
    The library doesn't check for overflows. You may also use this function
    for calculating signed values, but then you'll have to interpret the
    destination value yourself. For example, if you add 2 to a value
    of $FFFFFFFF FFFFFFFF, then the result will be $00000000 00000001.

SEE ALSO

            FAMESub64()
```

1.2 FAME.library/FAMEAllocObject

```
            NAME
    FAMEAllocObject -- Allocate a FAME Object

SYNOPSIS
    Object = FAMEAllocObject(Type)
      D0                     D0

    APTR FAMEAllocObject(ULONG)

FUNCTION
    Allocates and initializes a FAME structure for you, e.g. a struct
    FAMEDoorMessage. Initialized means that, for example, an allocated
    FAMEDoorMessage object comes with correctly filled System data
    (e.g. MN_SIZE).

    Using FAME structures is a bit different to other BBS systems.
    All public FAME structures must be allocated using this function.
    The advantage is that all of these structures may be expanded
    with future FAME versions while staying fully downward compatible
    with any older door program.
    In consequence, FIM doors always have to open FAME.library and
    allocate their FAMEDoorMessage via FAMEAllocObject(FOBJ_FAMEDoorMsg),
    before MC_DoorStart may be sent using the allocated FAMEDoorMessage.
    You must use FAMEFreeObject() to free all FAME Objects. Don't try
    to free it yourself if you want to stay upward compatible.

INPUTS
    Type            - The Object type you need (e.g. FOBJ_FAMEDoorMsg)

RESULT
    Object          - Address of the FAME Object or NULL if an error
                      occured. You may use DOS/IoErr() to see what's
                      been wrong. The error codes are defined in
                      include/libraries/FAME.x. The error code is set
                      to ERROR_NO_FREE_STORE if the allocation itself
                      failed due to lack of free memory.
```

```
SEE ALSO

              FAMEFreeObject()
```

1.3 FAME.library/FAMEAllocPooled

```
                NAME
      FAMEAllocPooled -- Allocate memory from your own pool

SYNOPSIS
      Memory = FAMEAllocPooled(byteSize, memAttrs, FAMEMemPool)
        D0                      D0        D1         A0

      APTR FAMEAllocPooled(ULONG, ULONG, APTR)

FUNCTION
      Allocate memory from your own memory pool. The pool functions are
      written for developers who want to keep their programs compatible
      with AmigaOS versions lower than 3.0, and it's also a good replace-
      ment because of some future-planned extra features. Stay tuned. :^)
      After creating a memory pool, you may allocate a memory block of
      the pool by passing the needed size and memAttrs and the pool where
      to take the memory from. But note that using memory pools is only
      useful if your program really does lots of allocations and deallo-
      cations, for example, if you need to allocate list elements or
      such. The pool and puddle size should be at least 20 to 100 times
      bigger than the allocations you do later, in order not to allocate
      new child pools so often. For allocating smaller things, like
      50 or 700 bytes (or such), pool/puddle sizes of 30K (or bigger !)
      are okay. So-called "tresh" sizes aren't needed with FAME pools.

INPUTS
      byteSize        - Number of bytes to allocate; equals AllocMem()

      memAttrs        - Currently, only MEMF_CLEAR is supported.

      FAMEMemPool     - The memory pool you allocated using
              FAMECreatePool()
                RESULT
      Memory          - Address of the requested memory block or NULL
                        if out of memory; equal to AllocMem(). You can
                        run out of memory if the needed size is bigger than
                        the remaining main pool space and also bigger than
                        the PuddleSize you have set up when creating the pool,
                        or if there is not enough free memory left to allo-
                        cate a new child pool.
                        Note that all allocated FAME pool memory blocks are
                        QuadWord (8-byte) aligned, which is very handy for
                        DMA transfers on systems equipped with a 68060 CPU.
NOTES
      Not-so-important technical note: every allocation needs 16 additional
      bytes for MemList linking and for a very high operation speed. That
      means that any free memory block (found in a pool by FAME.library)
      must be at least RequestedSize+16 bytes in size.
      Note this if you manage e.g. pools of 500000 bytes, allocating 20000
```

```
      bytes each call. This would allow only 24 instead of 25 allocations
      before the pool is full (no more memory blocks of 20000 bytes left).
      If you're using such very large memory allocations, use the following
      formula to calculate the needed pool and puddle sizes:

      NeededPoolSize = (NumberOfAllocationsToFit * (AllocationSize+16))

      However, this complex thing is only important if your later memory
      allocations have always the *same* and very large size. With smaller
      sizes (e.g. 1000 bytes), it doesn't matter if (e.g.) 900 bytes of a
      pool stay unused; with different MemSizes, there is absolutely no
      way to ensure that a pool gets always filled up to the very last byte.
      For almost all normal cases, this point has no weight anyway. :^)

SEE ALSO

              FAMECreatePool()
              ,
              FAMEDeletePool()
              ,
              FAMEFreePooled()
              ,
              FAMEResetPool()
```

1.4 FAME.library/FAMEAvailExe

```
NAME
      FAMEAvailExe -- Check if Name is an available program

SYNOPSIS
      Result = FAMEAvailExe(Name)
      D0                    D0

      LONG FAMEAvailExe(STRPTR Name)

FUNCTION
      Checks if Name is an available program, meaning that it can be
      found either resident or at the specified path. If Name doesn't
      contain a path part, the current directory is examined for the
      file.

INPUTS
      Name       - Standard AmigaDOS file name. This name may include
                   a path. FAMEAvailExe checks the resident segment
                   list for matches. If it wasn't found, FAME.library
                   tries to Lock Name. If successful, it checks if the
                   lock is really a file, and if the file is executable.

RESULT
      Result     - If NULL (FAE_NOMATCH), Name is neither resident nor
                   could it be locked in the given path anyway.
                   Possible results are:

      FAE_RESIDENT    - Program was found in the segment list (is loaded
                        resident).
```

```
      FAE_RESIDENTSYS - Program was found in the system segment list.

      FAE_LOADFILE    - File was found in the named location and is
                        executable.

      FAE_NOMATCH     - NULL; nothing found anyway.

      FAE_DATAFILE    - File was found, but has no HUNK_HEADER id.

      FAE_NOEBIT      - File was found and has a hunk header, but it's
                        executable protection bit isn't set.

      FAE_DIRECTORY   - Name is a directory, not a file.

      FAE_ERROR       - An I/O error occured while examining/opening/reading
                        the file. Use IoErr() to see what happened.
                        This Result tells you that Name is not loaded
                        resident, and that a dir entry with that name
                        exists.

NOTES
      Result is positive if Name can be launched as a program.
      Passing NULL or an empty string is harmless.
      Beyond AmigaDOS calls, this function uses 296 bytes of stack space.
```

1.5 FAME.library/FAMEChrCut

```
               NAME
      FAMEChrCut -- cut a string from a search char position

SYNOPSIS
      FAMEChrCut(String, CutChar, MaxSearchRange)
                  A0       A1          D0

      STRPTR FAMEChrCut(STRPTR, UBYTE, ULONG)

FUNCTION
      Cut a string from the first position of a specified char.
      The CutChar will be searched inside the source string; if
      a match was found, the right part of the source string gets cut
      at the position where the CutChar was found. If nothing was found,
      nothing gets changed. This function is not case-sensitive;
      german umlauts and other international chars are not considered.

INPUTS
      String          - The string to truncate
      CutChar         - The char to search for
      MaxSearchRange  - Important -the maximum string length to search for
                        the Char. Normally, this is SizeOf(String).

RESULT (V2)
      Result points to the (new) end position of the passed string, no
      matter if the CutChar was found or not. Result will point exactly
      to the trailing NULL byte of the string.
```

```
SEE ALSO

               FAMEChrCutCase()
              ,
               FAMEStrCut()
              ,
               FAMEStrCutCase()
```

1.6 FAME.library/FAMEChrCutCase

```
                NAME
      FAMEChrCutCase -- cut a string from a search char position

SYNOPSIS
      FAMEChrCutCase(String, CutChar, MaxSearchRange)
                      A0      A1           D0

      STRPTR FAMEChrCutCase(STRPTR, UBYTE, ULONG)

FUNCTION
      Cut a string from the first position of a specified char.
      The CutChar will be searched inside the source string; if
      a match was found, the right part of the source string gets cut
      at the position where the CutChar was found. If nothing was found,
      nothing gets changed. This function is similar to FAMEChrCut(),
      but case-sensitive, and faster.

INPUTS
      String          - The string to truncate
      CutChar         - The char to search for
      MaxSearchRange  - Important -the maximum string length to search for
                        the Char. Normally, this is SizeOf(String).

RESULT (V2)
      Result points to the (new) end position of the passed string, no
      matter if the CutChar was found or not. Result will point exactly
      to the trailing NULL byte of the string.

SEE ALSO

               FAMEChrCut()
              ,
               FAMEStrCut()
              ,
               FAMEStrCutCase()
```

1.7 FAME.library/FAMECreatePool

```
                NAME
      FAMECreatePool -- Create a private memory pool
```

```
SYNOPSIS
      FAMEMemPool = FAMECreatePool(poolSize, puddleSize, memAttrs, tags)
      D0                           D0         D1         D2      D3

      APTR FAMECreatePool(ULONG poolSize, ULONG puddleSize, ULONG memAttrs,  ↩
         struct TagItem *)

FUNCTION
      Create an own memory pool. Refer to the
              FAMEAllocPooled()
               section
      of this manual. Own memory pools are very useful if you often allocate
      e.g. list elements for bigger lists etc. This reduces memory fragmen-
      tation and is also handled much faster, since the memory lists
      don't contain hundreds or thousands of MemEntries which belong to
      other tasks. In addition, FAME.library won't ever protect the pool
      memory lists using Forbid(). In future versions, public semaphore-
      protected pools may also be possible, where, for example, a task
      may allocate an Exec Message from a private pool of the receiver
      task.

INPUTS
      poolSize        - The size of the main memory pool. If you try to
                        FAMEAllocPooled() more bytes than currently free, the
                        library will create a "child" pool with a so-called
                        "puddle" size you specified. PoolSize must be at
                        least 1024 bytes.
      puddleSize      - Similar to poolSize. If the remaining main memory
                        pool cannot hold the requested size, a new child
                        pool gets allocated and attached to your main pool.
                        Then, the needed memory gets taken from the new
                        child pool instead. You may pass NULL for PuddleSize;
                        FAMEAllocPooled() will fail then instead of attaching
                        new child pools if the main pool is full.

      memAttrs        - For allocating pools and child pools, the following
                        Exec memory attributes are supported:

                        - MEMF_CHIP
                        - MEMF_FAST
                        - MEMF_24BITDMA

                        MEMF_CLEAR is specified with
              FAMEAllocPooled()
              .

      Tags            - Currently unused. Pass NULL or an empty TagList
                        (pointing to TAG_END).

RESULT
      FAMEMemPool     - A FAME internal memory pool structure to be passed
                        to some of the other pool functions, or NULL if
                        the allocation failed.

SEE ALSO

              FAMEAllocPooled()
```

```
,
FAMEDeletePool()
,
FAMEFreePooled()
,
FAMEResetPool()
```

1.8 FAME.library/FAMECutANSI

```
NAME
      FAMECutANSI -- Cut ANSI control sequences off a string (V4).

SYNOPSIS
      FAMECutANSI(String, Flags)
                  A0     D0

      STRPTR FAMECutANSI(STRPTR, ULONG)

FUNCTION
      This function strips standard ANSI control codes off a given string.

INPUTS
      String      - The string to convert.
      Flags       - The passed flags control which kind of ANSI commands
                    you want to have stripped off. All cut options regard
                    both ESC and CSI introduced command sequences.
                    Currently, only one type is supported:

                    FCAF_STYLECMDS

                     - this option will remove all ANSI sequences which
                       control output colours and text style-formatting;
                       namely: every Escape sequence ending with a lower-
                       case "m" char.

RESULT
      Points to the passed, converted string (same as input).
```

1.9 FAME.library/FAMEDeletePool

```
               NAME
      FAMEDeletePool -- Remove and deallocate a memory pool

SYNOPSIS
      FAMEDeletePool(FAMEMemPool)
                     A1

      VOID FAMEDeletePool(APTR)

FUNCTION
      Removes a pool together with all it's child pools. Must be called
      before finishing your program in order to undo FAMECreatePool().
```

```
      All allocated memory areas belonging to this pool get invalid
      and don't need to be freed anymore. This is the fast way to
      free everything by one single call.

INPUTS
      FAMEMemPool     - the structure you got from
              FAMECreatePool()
              .

SEE ALSO

              FAMEAllocPooled()
              ,
              FAMECreatePool()
              ,
              FAMEFreePooled()
              ,
              FAMEResetPool()
```

1.10 FAME.library/FAMEExecuteDir

```
               NAME
      FAMEExecuteDir -- Run all programs in a given directory

SYNOPSIS
      Result = FAMEExecuteDir(DirLock, Tags, Args)
      D0                      D0      A0    D1

      APTR FAMEExecuteDir(BPTR DirLock, struct TagItem *, STRPTR Args)

FUNCTION
      This function examines the given directory and starts all programs
      found in that dir. If a program was successfully started, it's name
      and ReturnCode gets added to the FAMEExecuteDirList structure you'll
      receive as result of this function. This list contains the names and
      DOS ReturnCodes of all successfully started programs.
      If you're using the SYS_Asynch Tag, the ReturnCodes are not available
      and will be NULL then.
      FAMEExecuteDir() uses DOS/SystemTagList() to launch the programs.

INPUTS
      DirLock    - Lock of the directory to use. This Lock *must* be a
                   SHARED Lock ! Otherwise, this function can't find
                   the name of this Lock. This is a dos.library failure;
                   just try NameFromLock() with an EXCLUSIVE directory
                   Lock and you'll see what i mean.
      Tags       - DOS Tags for SystemTagList().
      Args       - Command arguments (optional -you may pass NULL).

RESULT
      Result     - FAMEExecuteDirList. The result for this function is
                   not easy to explain. Because of the many possible
                   errors, FAMEExecuteDir() will not fail if a program
                   wasn't succesfully started (e.g. no memory for program
                   hunks). Instead, the directory gets examined for further
```

```
                    executable files until the directory contains no more
                    entries.
                    This function combines Result and IoErr() when returning.
                    There are four possible things you may get:

                    Result <> NULL, IoErr = NULL:
                       Everything went okay, but there may be some
                       program(s) which weren't successfully launched.

                    Result = NULL, IoErr = NULL:
                       Everything okay, but the directory does not contain
                       any executable files (no files, no list).

                    Result = NULL, IoErr <> NULL:
                       Fatal error (for example, directory had a read error).
                       Nothing was executed.

                    Result <> NULL, IoErr <> NULL:
                       Fatal error, but some programs have been successfully
                       executed.

                    Anyway, if the Result was non-zero, it will be a valid
                    FAMEExecuteDirList structure which must be freed using

                FAMEFreeExecuteDirList()
                .

NOTES
      This function also executes all CLI scripts it finds in the
      given directory (script bit must be set), but all of tse scripts
      must not have any spaces in their complete path and file name,
      because SystemTagList() doesn't handle quotes with script path/file-
      names correctly, while executable files are started within or
      without quotes (V39).

      Beyond AmigaDOS calls, this function uses approx. 1200 bytes of
      stack space.

SEE ALSO

                FAMEFreeExecuteDirList()
```

1.11 FAME.library/FAMEFileCopy

```
                NAME
      FAMEFileCopy -- Copy a file to another destination

SYNOPSIS
      Result = FAMEFileCopy(SourceFH, DestFH, SrcSize, MaxMem)
      D0                    D0       D1       D2      D3

      LONG FAMEFileCopy(BPTR SourceFH, BPTR DestFH, ULONG SrcSize, ULONG MaxMem)

FUNCTION
      Copy a file to another destination. You may specify a maximum copy
```

```
      buffer size, and the number of bytes to copy. If you set SrcSize to
      -1, the whole file gets copied. The passed file handles don't get
      seeked to their start positions before copying is performed.

INPUTS
      SourceFH    - Filehandle you wish to copy from
      DestFH      - Filehandle you wish to copy to
      SrcSize     - Number of bytes to be copied, or -1 for all until EOF.
      MaxMem      - Maximum copy buffer

RESULT
      Result      - NULL if everything went okay; otherwise, you may see
                    what's been wrong using dos/IoErr(). FAMEFileCopy()
                    returns standard dos.library errors.

NOTE
      For copying files from DOS file names, refer to
              FAMEDosMove()
              .
```

1.12 FAME.library/FAMEFillMem

```
                NAME
      FAMEFillMem -- fill a larger memory area with a byte value

SYNOPSIS
      FAMEFillMem(Buffer, FillByte, Size)
                    A0       D0      D1

      VOID FAMEFillMem(APTR, UBYTE, LONG)

FUNCTION
      Fill-up a buffer with a special byte value.
      This function is equal to
              FAMEMemSet()
              , but highly optimized for
      bigger memory areas. Use it for filling at least 208 bytes of
      memory; with lower sizes,
              FAMEMemSet()
               works faster.

INPUTS
      Buffer          - The buffer to be filled. This address MUST be WORD
                        aligned ! With 68020+ CPUs, FAMEFillMem() gets
                        faster if a LONG aligned buffer is used.
      FillByte        - The byte value to fill with
      Size            - How many bytes to write. This value is limited
                        to a maximum of 13,631,280 bytes. If you pass a
                        higher value, nothing will happen.

SEE ALSO

               FAMEMemSet()
              ,
```

```
                FAMEStrFil()
```

1.13 FAME.library/FAMEFreeDevInfoList

```
NAME
      FAMEFreeDevInfoList -- free a FAMEDevInfoList

SYNOPSIS
      FAMEFreeDevInfoList(FAMEDevInfoList)
                                A1

      VOID FAMEFreeDevInfoList(APTR)

FUNCTION
      Frees a FAMEDevInfoList.

SEE ALSO
      FAMEGetDevInfoList()
```

1.14 FAME.library/FAMEFreeDiskSpace

```
NAME
      FAMEFreeDiskSpace -- returns how many bytes are free on a volume.

SYNOPSIS
      FAMEFreeDiskSpace(Name)
                         A0

      LONG FAMEFreeDiskSpace(STRPTR)

FUNCTION
      Returns the amount of free bytes on the disk where the given file or
      drawer belongs to. You may specify any kinda AmigaDOS file or directory
      name. This string gets directly passed to DOS/Lock(). If the Lock()
      call was successful, the function attempts to return the amount of
      free disk space belonging to the Volume where the locked object resides.
      For example, you may pass "DH0:"; or you may use a filename, like
      "MyData.Big", where the root (the volume) of the current directory
      gets examined for the number of free bytes.
      By the way: the flexibility of this function is a dos.library feature.

INPUTS
      Name    - A standard DOS file name.

RESULT
      Amount  - The amount of free bytes, or -1 if an error occured.
                You may use IoErr() to examine the fault.
```

1.15 FAME.library/FAMEFreeExecuteDirList

```
               NAME
      FAMEFreeExecuteDirList -- free a FAMEExecuteDirList

SYNOPSIS
      FAMEFreeExecuteDirList(FAMEExecuteDirList)
                                      A1

      VOID FAMEFreeExecuteDirList(APTR)

FUNCTION
      Frees a FAMEExecuteDirList.

INPUTS
      FAMEExecuteDirList - The result of a previous FAMEExecuteDir() call.

SEE ALSO

              FAMEExecuteDir()
```

1.16 FAME.library/FAMEFreeFile

```
               NAME
      FAMEFreeFile -- free a FAME File loaded using
              FAMELoadFile()
              .

SYNOPSIS
      FAMEFreeFile(FAMEFile)
                   A1

      VOID FAMEFreeFile(APTR)

FUNCTION
      Free a FAME File.

INPUTS
      Object          - The FAME File to free.

NOTE
      This function does not close the file's FileHandle, if it's still
      open. If you have specified FLFF_KEEPFH with
              FAMELoadFile()
              , you'll
      have to Close() it yourself before calling FAMEFreeFile().

SEE ALSO

              FAMELoadFile()
```

1.17 FAME.library/FAMEFreeObject

```
               NAME
      FAMEFreeObject -- free a FAME Object.

SYNOPSIS
      FAMEFreeObject(Object)
                     A1

      VOID FAMEFreeObject(APTR)

FUNCTION
      Free a FAME Object.

INPUTS
      Object          - The FAME Object to free.

SEE ALSO

              FAMEAllocObject()
```

1.18 FAME.library/FAMEFreePooled

```
               NAME
      FAMEFreePooled -- free a pool memory area.

SYNOPSIS
      FAMEFreePooled(memory)
                     A1

      VOID FAMEFreePooled(APTR)

FUNCTION
      Free a pool memory area. The memory gets put back to the pool it was
      taken from. If the memory area has any free (unused) neighbours,
      theyl get linked together to one large memory block again, which
      will reduce pool memory fragmentation. If this was the last memory
      area in the current child pool, the whole child pool gets unlinked
      and deallocated, too.

INPUTS
      memory          - Pointer to the memory you got from
              FAMEAllocPooled()
              .
                        Do not free one memory area twice -the result may
                        be a guru.

NOTE
      In contrast to the exec.library function, you don't need to pass
      any further arguments except Memory.

SEE ALSO

              FAMEAllocPooled()
              ,
              FAMECreatePool()
```

```
,
FAMEDeletePool()
,
FAMEResetPool()
```

1.19 FAME.library/FAMEFSearch

```
NAME
      FAMEFSearch -- Fast search for a match in a file with continous search

SYNOPSIS
      Result = FAMEFSearch(SearchString, SearchFH)
      D0                        A0          D0

      LONG FAMEFSearch(STRPTR SearchString, BPTR SearchFH)

FUNCTION
      This function searches a string inside a file.
      Because of speed reasons, the file buffer is set to 100000 bytes.
      This function is *not* case-sensitive, so it'll find all matches,
      no matter which case. German umlauts and other language specific
      chars are not considered, so that case matters then.

INPUTS
      SearchString - String to search for.
      SearchFH     - Filehandle of the file where to search the string.

RESULT
      Result       - The *absolute* position of the string in the file,
                     no matter what seek position the FH had at the
                     beginning of the function call. In addition, the
                     seek position is set directly to the string's start
                     position; so you may call this function multiple times
                     to find all contents by simply seeking one byte forward
                     and calling FAMEFSearch() again.
                     If an error occured, or if the string wasn't found,
                     you'll receive -1 as result. Call dos/IoErr() to
                     get the exact error code. IoErr() will return NULL if
                     the string wasn't found. If a real problem occured,
                     you'll get a standard DOS error code. This may be
                     ERROR_NO_FREE_STORE, for example, or a read error.
```

1.20 FAME.library/FAMEGetDevInfoList

```
               NAME
      FAMEGetDevInfoList -- get infos about connected devices

SYNOPSIS
      FAMEGetDevInfoList()

      APTR FAMEGetDevInfoList(VOID)
```

```
FUNCTION
      Returns a single linked, sorted FAMEDevInfoList of device information
      data. Refer to libraries/FAME.h/.i for detailed information.
      After examining all data, this list must be freed using the

              FAMEFreeDevInfoList()
               function.

SULT
      List    - FAMEDevInfoList or NULL if an error occured.
                On error, you may use IoErr() to see what happened.

SEE ALSO

              FAMEFreeDevInfoList()
                BUGS
      Currently, the FAMEDevInfoList doesn't get sorted. Still to do.
```

1.21 FAME.library/FAMEIsNumStr

```
NAME
      FAMEIsNumStr -- check if a string contains numeric digits only.

SYNOPSIS
      FAMEIsNumStr(String)
                   A0

      ULONG FAMEIsNumStr(STRPTR String)

FUNCTION
      Checks if the given string only contains chars from "0" to "9".

INPUTS
      String - the string to check

RESULT
      Result  - NULL (false) if the string contains any non-numeric digits.
```

1.22 FAME.library/FAMELoadFile

```
                NAME
      FAMELoadFile -- easily load a file into memory.

SYNOPSIS
      FAMELoadFile(Name, MemAttr, MaxSize, Flags)
                   D0      D1       D2      D3

      APTR FAMELoadFile(STRPTR Name, ULONG MemAttr, ULONG MaxSize, ULONG Flags)

FUNCTION
      With this function you may easily load a file into memory with one
```

```
      single function call. No more need to handle the many errors which
      may occur with Lock(), Open(), Examine(), AllocVec(), Read(), wrong
      file types etc.

INPUTS
      Name            - standard AmigaDOS filename.
      MemAttr         - exec memory attributes to use for the file buffer.
      MaxSize         - maximum file size allowed. If you set this to NULL,
                        FAME.library will try to load the file anyway.
                        This argument may be useful for writing e.g. BBS
                        door programs, which (normally) shouldn't use more
                        than about 512K of memory to make sure not to hang-up
                        a BBS due to lack of memory (a poorly written door
                        running on another node may cause huge problems in
                        out-of-memory situations).
      Flags           - aditional flags as follows:

                        FLFF_KEEPFH
                          Don't close the file after loading.
                          If this flag is set, ffil_FH gets filled with
                          the file's FileHandle; otherwise this field
                          will be empty.
                          Useful to prevent other tasks from accessing the
                          file while you may want to examine it's FileHandle
                          or if you're going to do any changes to the file.
                          The FH's seek position points to EOF, and the FH
                          itself is of MODE_OLDFILE. Use ChangeMode() to
                          change to another access mode.

RESULT
      FAMEFile        - A FAMEFile structure or NULL if an error occured.
                        This may be any typical DOS Error Code, or, in
                        addition, it may be ERROR_NO_FREE_STORE if you
                        run out of memory, or ERROR_OBJECT_TOO_LARGE if
                        the file is bigger than the specified MaxSize.
                        Use IoErr() to see wt happened.
NOTES
      Result is also valid if the file size was NULL.

      Beyond AmigaDOS I/O, this function uses 296 bytes of Stack space.

SEE ALSO

               FAMEFreeFile()
```

1.23 FAME.library/FAMELoadFilePooled

```
                NAME
      FAMELoadFilePooled -- easily load a file into memory using a pool.

SYNOPSIS
      FAMELoadFilePooled(Name, MemAttr, MaxSize, Flags, FAMEMemPool)
                           D0      D1      D2       D3        a0
```

```
      APTR FAMELoadFilePooled(STRPTR Name, ULONG MemAttr, ULONG MaxSize, ULONG  ↩
         Flags, APTR FAMEMemPool)

FUNCTION
      With this function you may easily load a file into memory with one
      single function call. No more need to handle the many errors which
      may occur with Lock(), Open(), Examine(), AllocVec(), Read(), wrong
      file types etc.
      In contrast to
              FAMELoadFile()
              , this function allocates the needed
      memory from your own memory pool. Also, the resulting structure
      (struct FAMEPoolFile) is different to the FAMEFile structure returned
      by
              FAMELoadFile()
              . Here you have got two additional elements at the top
      of the structure, which are initialized to NULL and may be freely used
      by your program, for example, to link several Pooled Files together
      to a single- or double-linked list.

INPUTS
      Name            - standard AmigaDOS filename.
      MemAttr         - exec memory attributes to use for the file buffer,
                        ass supported by
              FAMEAllocPooled()
              .
      MaxSize         - maximum file size allowed. If you set this to NULL,
                        FAME.library will try to load the file anyway.
                        This argument may be useful for writing e.g. BBS
                        door programs, which (normally) shouldn't use more
                        than about 512K of memory to make sure not to hang-up
                        a BBS due to lack of memory (a poorly written door
                        running on another node may cause huge problems in
                        out-of-memory situations).

                        Hint: MaxSize should not be bigger than the Pool's
                        PuddleSize.

      Flags           - aditional flags as follows:

                      FLFF_KEEPFH
                        Don't close the file after loading.
                        If this flag is set, fpof_FH gets filled with
                        the file's FileHandle; otherwise this field
                        will be empty.
                        Useful to prevent other tasks from accessing the
                        file while you may want to examine it's FileHandle
                        or if you're going to do any changes to the file.
                        The FH's seek position points to EOF, and the FH
                        itself is of MODE_OLDFILE. Use ChangeMode() to
                        change to another access mode.

      FAMEMemPool     - a FAME Memory Pool where to take the needed
                        memory. Refer to FAMECreatePool() for more
                        details.

RESULT
```

```
      FAMEPoolFile    - A FAMEPoolFile structure or NULL if an error
                        occured. This may be any typical DOS Error Code,
                        or, in addition, it may be ERROR_NO_FREE_STORE
                        if you run out of memory, or ERROR_OBJECT_TOO_LARGE
                        if the file is bigger than the specified MaxSize.
                        Use IoErr() to see what happened.
NOTES
      Result is also valid if the file size was NULL.

      Freeing a FAMEPoolFile is done by a
              FAMEFreePooled()
               call, or
      by calling
              FAMEDeletePool()
               which frees the Pool and all loaded
      files with one single call. Do not try to free a FAMEPoolFile using
      FAMEFreeFile() !

      Beyond AmigaDOS I/O, this function uses 296 bytes of Stack space.
```

1.24 FAME.library/FAMEMemSet

```
                NAME
      FAMEMemSet -- Fill a buffer with a char

SYNOPSIS
      FAMEMemSet(FillBuffer, FillChar, NumberOfChars)
                      A0        D0           D1

      VOID FAMEMemSet(APTR FillBuffer, UBYTE FillChar, WORD NumberOfChars)

FUNCTION
      Fill up the buffer with a char up to NumberOfChars. This function
      is equal to
              FAMEStrFil()
              , but doesn't add a null byte at the end.
      As you see, many functions are somewhat trivial, but C programmers may
      have use of it while going without the ANSI C functions, making the
      executables much smaller.
      Note that if you want to fill large buffers, you should use

              FAMEFillMem()
               instead, because it's highly optimizd for bigger
      buffers.

INPUTS
      FillBuffer    - The buffer to fill up.
      FillChar      - The fill character.
      NumberOfChars - The number of characters to write (WORD size !).

SEE ALSO

               FAMEFillMem()
              ,
```

```
                FAMEStrFil()
```

1.25 FAME.library/FAMEDosMove

```
NAME
      FAMEDosMove -- move a file from one directory to another

SYNOPSIS
      FAMEDosMove(Src, Dest, MaxBuf, Flags)
                  D0    D1     D2     D3

      ULONG FAMEDosMove(STRPTR Src, STRPTR Dest, ULONG MaxBuf, ULONG Flags)

FUNCTION
      Move (or copy) the source file to the given destination, which may
      be a directoy or a full (new) file name. When moving, FAMEDosMove()
      first tries to rename.

INPUTS
      Src     - Source file with full path.
      Dest    - Destination path, may include the new file name itself.
      MaxBuf  - Maximum copy buffer size if file must be copied. Try
                using a buffer of at least 256-512K in order to perform
                high-speed copies.

      Flags   - FDMF_NODELETE (V2)

                - do not delete source (copy instead of moving)

              - FDMF_KEEPDATA (V2)

                - if the file wasn't moved by renaming, FAMEDosMove() will
                  keep the source file comment and protection bits (except
                  the archive bit).

RESULT
      Result  - Success (Boolean). On failure, IoErr() helps to find out
                about what went wrong.

NOTES
      Beyond AmigaDOS calls, this function uses 832 bytes of stack space.

      Currently, this function does not copy directories anyway.
```

1.26 FAME.library/FAMENum64ToStr

```
                NAME
      FAMENum64ToStr -- convert a 64-bit value to a decimal string (V3)

SYNOPSIS
      FAMENum64ToStr(ValueHi, ValueLo, Flags, BufSize, Buffer)
                     d0        d1     d2       d3      a0
```

```
      LONG FAMENum64ToStr(ULONG, ULONG, ULONG, ULONG, STRPTR)

FUNCTION
      This function converts a 64-bit numeric value to a decimal string.
      FAMENum64ToStr() is compatible to
              FAMENumToStr()
              , except that the
      value to convert consists of two longwords, and hex or binary
      conversion is not included since 64-bit hex/binary strings may be
      created by two
              FAMENumToStr()
               calls.

INPUTS
      ValueHi - The upper 32 bits of the value to convert
      ValueLo - The lower 32 bits of the value to convert
      BufSize - The maximum output buffer size.
                If BufSize is NULL, FAME.library assumes that the output
                buffer is big enough anyway. Refer to "Table Of Buffer
                Sizes" to see how big the destination string buffer should
                be. The maximum needed buffer size is 29 bytes including
                the trailing NULL byte.
                If the output string doesn't fit into the Buffer, RESULT
                will be -1 and the output buffer won't get touched anyway.
                It's up to you about what to do if the string doesn't fit
                into the buffer, e.g. using strings like ">1 Bio." or such.

      Flags   - Options as follows. All flags have the same names as the
                ones used for
              FAMENumToStr()
              .

      FNSF_LEADINGZEROES - Output string with leading zeroes. This option
                           overrides the text formatting flags.

      FNSF_LEADINGSIGN   - Add a leading arithmetic symbol ("#").
                           This symbol is added as a prefix, placed directly
                           left of the leftmost digit.
                           Remember that this option raises the needed output
                           buffer size. See "Table Of Buffer Sizes" below.

      FNSF_RIGHTFORMAT   - String gets right-formatted. The number of
                           leading spaces depends on the buffer size used.
                           Refer to "Table Of Buffer Sizes" below.

      FNSF_LEFTFORMAT    - Buffer gets filled up with spaces behind the
                           numeric string. Refer to "Table Of Buffer
                           Sizes" below.

      FNSF_CENTERFORMAT  - The numeric string gets centered inside the
                           output buffer. Refer to "Table Of Buffer Sizes"
                           below.

      FNSF_TENDLEFT      - This flag works together with FNSF_CENTERFORMAT.
                           Normally, this function tends to place the center
                           formatted strings a bit more right, because half
```

space chars cannot be printed. With this flag you may change this behaviour to put the strings more left.

FNSF_GROUPING - Add group separators to the string. Example: Output "42658313" as "42,658,313". Remember that this option raises the needed output buffer size. See "Table Of Buffer Sizes" below.

FNSF_NUMLOCALE - If the FNSF_GROUPING flag is set, with this option the current Locale numeric group separator is used instead of the default group separator char.

FNSF_MONLOCALE - If the FNSF_GROUPING flag is set, with this Flag the current Locale monetary group separator is used instead of the default group separator char.

FNSF_SIGNED - Interpret Value as signed number. BufSize always raises by one additional byte. Negative numbers get a "-" prefix, positive ones get a " " prefix.

FNSF_PLUSSIGN - If FNSF_SIGNED is set, positive numbers get a "+" sign added, instead of the default " " char.

FNSF_SWAPSIGNS - If both FNSF_SIGNED and FNSF_LEADINGSIGN flags are set, this option swaps the two chars from "-#" to "#-".

RESULT

Result - NULL if everything went ok, or -1 if the string didn't fit into the buffer (this error may only occur if BufSize was non-zero).

NOTES

Same rules as with FAMENumToStr().

Table Of Buffer Sizes

The following table shows up the minimum output buffer sizes needed to convert any Value without the event of a buffer overflow. This is useful if you want position-formatted strings.

All buffer sizes include the trailing NULL byte.

Type	Pure	With Leading Symbol	With Separators	With Both
LONG	22	23	28	29
ULONG	21	22	27	28

SEE ALSO

Locale Notes

```
,
FAMENumToStr()
.
```

1.27 FAME.library/FAMENumToStr

```
            NAME
      FAMENumToStr -- convert a numeric value to a string

SYNOPSIS
      FAMENumToStr(Value, Flags, BufSize, Buffer)
                   d0      d1     d2       a0

      LONG FAMENumToStr(ULONG, ULONG, ULONG, STRPTR)

FUNCTION
      This function converts a numeric value to either a decimal, hex,
      or binary string. This function is useful in situations where a
      standalone AmigaOS FormatString isn't flexible enough.
      The converted string is terminated by a NULL byte. Keep this in
      mind -never set BufSize to 1 ! It must always be either NULL or
      greater than 1 !

INPUTS
      Value   - The numeric value to convert
      BufSize - The maximum output buffer size.
                If BufSize is NULL, FAME.library assumes that the output
                buffer is big enough anyway. Refer to "Table Of Buffer
                Sizes" below to see how many bytes are needed.
                If the output string doesn't fit into the Buffer, RESULT
                will be -1 and the output buffer won't get touched anyway.
                It's up to you about what to do if the string doesn't fit
                into the buffer, e.g. using strings like ">1 Mio." or such.

      Flags   - Options as follows:
                (same usage as, for example, the exec memory attributes)

      FNSF_HEX           - Convert Value to hex string (default: decimal)

      FNSF_BIN           - Convert Value to binary string (default: decimal)

      FNSF_WORD          - Value has WORD size (default: LONG)

      FNSF_BYTE          - Value has BYTE size (default: LONG)

      FNSF_LEADINGZEROES - Output string with leading zeroes. Of course,
                           this option overrides the text formatting flags.

      FNSF_LEADINGSIGN   - Add a leading arithmetic symbol, depending
                           on the stated numeric system:
                           - Decimal strings get a "#" prefix
                           - Hex strings get a "$" prefix
                           - Binary strings get a "%" prefix
                           The prefix is always placed directly left of the
                           leftmost digit.
```

```
                        Remember that this option raises the needed output
                        buffer size. See "Table Of Buffer Sizes" below.

      FNSF_RIGHTFORMAT  - String gets right-formatted. The number of
                        leading spaces depends on the numeric system,
                        the Value size (BYTE/WORD/LONG) and the buffer
                        size. Refer to "Table Of Buffer Sizes" below.

      FNSF_LEFTFORMAT   - Buffer gets filled up with spaces behind the
                        numeric string. Refer to "Table Of Buffer
                        Sizes" below.

      FNSF_CENTERFORMAT - The numeric string gets centered inside the
                        output buffer. Refer to "Table Of Buffer Sizes"
                        below.

      FNSF_TENDLEFT     - This flag works together with FNSF_CENTERFORMAT.
                        Normally, this function tends to place the center
                        formatted strings a bit more right, because half
                        space chars cannot be printed. With this flag you
                        may change this behaviour to put the strings more
                        left.

      FNSF_LOWERCASE    - Output hex chars in lower case

      FNSF_GROUPING     - Add group separators to the string. Example:
                        Output (decimal) "42658313" as "42,658,313".
                        Remember that this option raises the needed output
                        buffer size. See "Table Of Buffer Sizes" below.

      FNSF_NUMLOCALE    - If the FNSF_GROUPING flag is set, with this option
                        the current Locale numeric group separator is used
                        instead of the default group separator char.
                        This option only affects decimal strings.

      FNSF_MONLOCALE    - If the FNSF_GROUPING flag is set, with this Flag
                        the current Locale monetary group separator is
                        used instead of the default group separator char.
                        This option only affects decimal strings.

      FNSF_SIGNED       - Interpret Value as signed number. BufSize always
                        raises by one additional byte. Negative numbers
                        get a "-" prefix, positive ones get a " " prefix.

      FNSF_PLUSSIGN     - If FNSF_SIGNED is set, positive numbers get a "+"
                        sign added, instead of the default " " char.

      FNSF_SWAPSIGNS    - If both FNSF_SIGNED and FNSF_LEADINGSIGN flags are
                        set, this option swaps the two chars from (e.g.)
                        "-$" to "$-".

RESULT
      Result  - NULL if everything went ok, or -1 if the string didn't fit
                into the buffer (this error may only occur if BufSize was
                non-zero).

NOTES
```

If you like system crashes, try setting undefined flags.. :^)
Also, do *not* use senseless flag combinations. For example, don't use FNSF_WORD together with FNSF_BYTE.

Formatting and leading zeroes: the number of spaces/zeroes depends on the given BufSize, but the whole string is never longer than the maximum needed number of chars. If you convert a decimal WORD value using a BufSize of 500 bytes, the string will only contain 5 digits, zeroes or spaces plus optional arithmetic sign and group separator, because the result cannot have more than 5 digits (the highest word value is 65535).

Grouping: beyond the group separator chars, other Locale Grouping preferences are ignored in order to keep the used buffer sizes fixed. Groups of decimal characters are always 3 chars in size. Groups are separated from the rightmost digit to the left side, e.g. "37,653", "1,986,221" or "298,344".
Hex strings get split into two equal groups.
Binary numbers get split into groups of 8 chars.
However, strings of BYTE size never contain any group separators, no matter which numeric system is used.
Default group separarator for decimal strings is the "," char.
For hexadecimal, a Space character (" ") is used; and the dot (".") is used with binary.
the FNSF_USExxxLOCALE flags have no effect with hex or binary strings.

Table Of Buffer Sizes

The following table shows up the minimum output buffer sizes needed to convert any Value without the event of a buffer overflow.

All buffer sizes include the trailing NULL byte.

String type		Pure	With Leading Symbol	With Separators	With Both
Decimal	LONG	11	12	14	15
	ULONG	12	13	15	16
	WORD	6	7	7	8
	UWORD	7	8	8	9
	BYTE	4	5	4	5
	UBYTE	5	6	5	6
Hex	LONG	9	10	10	11
	ULONG	10	11	11	12
	WORD	5	6	6	7
	UWORD	6	7	7	8
	BYTE	3	4	3	4
	UBYTE	4	5	4	5
Binary	LONG	33	34	36	37
	ULONG	33	34	36	37
	WORD	17	18	18	19
	UWORD	17	18	18	19
	BYTE	9	10	9	10
	UBYTE	9	10	9	10

```
      ------------------------------------------------------------------------

      Signed binary numbers don't need an additional byte, because the
      highest bit doesn't need to get displayed anymore; if you pass 255 as
      Value and FNSF_BINARY and FNSF_SIGNED as Flags, you'll get "-0000001"
      as result, instead of "11111111". Signed binary is rarely used..

      This function description may be somewhat confusing, all in all.
      I only suggest to try it out. Try several values in hex, binary
      or decimal format, and do not use any additional flags first. Then,
      add some flags (one after each other), and look what happens.
      In general, it may be helpful to know that "BufSize" is just a little
      bonus for easy text formatting. Let's have an example: you want to
      convert several (unknown) decimal values, which are always inside a
      range from 0 to 99999, and you want all output strings right-formatted,
      written into a box (or gadget) of 5 chars width. You'll have to use
      LONG conversion because 99999 is larger than a WORD. Now the library
      function will always insert at least five zeroes or spaces on the
      left side of the string, due to the fact that LONGs may have up to 10
      decimal digits. This could be too much. Now you may limit BufSize to
      5 (+ null byte = 6) bytes, and the problem is solved. But, whenever a
      Value is larger than 99999, a sixth digit was needed anyway. In this
      case, the buffer overflow error (ReturnCode = -1) would occur.
      But since you're sure about that the Value is never bigger than 99999,
      there should be no sweat about anyway.. :^)
      For most standard applications, simply setting BufSize to NULL while
      using a buffer of at least 37 bytes is a practical method.

      Hint: try combining FAMENumToStr() with RawDoFmt() by converting the
      needed values to size formatted strings which get inserted into a
      complete AmigaOS FormatString using the "%s" command. For example:
      "Address: %s, Offset: %s, File Size: %s, Year: %s", and so on (nice
      for data tables, statistics etc.).

SEE ALSO
      Locale
              Notes
              ,
              FAMENum64ToStr()
              .
```

1.28 FAME.library/FAMEOverallBytes

```
NAME
      FAMEOverallBytes -- get an overall amount of free harddisk space

SYNOPSIS
      FAMEOverallBytes(UlPathList, MinFreeMB)
                       A0          D0

      ULONG FAMEOverallBytes(APTR UlPathList, ULONG MinFreeMB)

FUNCTION
      This function scans a list of given directories for the overall
```

```
      amount of free bytes.

INPUTS
      UlPathList      - a FAMEUlPathList structure containing all pathes
                        to be examined. Each volume is only added once
                        to the result, meaning that several pathes
                        belonging to the same volume are not counted
                        several times.

      MinFreeMB       - a value you may specify, which gets subtracted from
                        every directory's free bytes amount. This value is
                        used by doors which handle a sysop-defined minimum
                        bytes which shall always be free in any upload path.

RESULT
      FreeBytes       - The overall amount of free bytes (-MinFreeMB) of all
                        given directories.
```

1.29 FAME.library/FAMEPostFile

```
NAME
      FAMEPostFile -- post an (uploaded) file to a FAME conference

SYNOPSIS
      FAMEPostFile(Name, ULPathList, MinFreeMB, BufSize, Flags)
                   D0        A0         D1        D2      D3

      APTR FAMEPostFile(STRPTR Name, APTR ULPathList, ULONG MinFreeMB, ULONG  ↩
         BufSize, ULONG Flags)

FUNCTION
      Post a file to a FAME BBS conference, where FAME.library moves (or
      copies) the file to one of a list of destination pathes as given as
      ULPathList argument. File comments and protection bits may be copied
      from the source file (V2).

INPUTS
      Name            - Full file name

      ULPathList      - a FAMEUlPathList structure which contains several
                        allowed upload directories, where FAME.library may
                        move the file to (FAME.library will use the first
                        suitable path it finds). The FAMEUlPathList structure
                        can be found in include/libraries/FAME.h(.i).
                        Passing a NULL pointer is harmless (V2).

      MinFreeMB       - The minimum Amount of MegaBytes available in the
                        destination directory wherever the file may go.
                        If the amount of free space plus src file size is
                        smaller than the passed MinFreeMB value, the actual
                        directory is skipped and FAME.library tries to use
                        the next one in the list. If nothing was found,
                        FAME.library returns NULL and sets IoErr to
                        ERROR_DISK_FULL. This argument counts in MegaBytes,
                        not in bytes.
```

BufSize - Maximum copy buffer size to use if the file can't be moved by renaming.

Flags - Several Flags, as there are:

FPFF_NODELETE

- Don't delete source file anyway. This flag forces FAME.library always to copy the file, even if it could simply be moved (renamed).

FPFF_REPLACE

- File gets always posted to the first suitable directory (with enough free space), no matter if a file with the same name already exists in any of the given directories. All existing files with same name get deleted if posting was successful (destination directory cleanup).
  If one of any additionally existing files coldn't be deleted, FAME.library will not fail, but it will set IoErr to ERROR_DIRECTORY_NOT_EMPTY in order to tell you that posting the file was completely successful; only the following directory cleanup failed.

FPFF_CHECKONLY

- Only test if the file already exists in any of the given directories; nothing gets written or deleted anyway.

FPFF_KEEPDATA (V2)

- if the file wasn't moved by renaming, FAMEDosMove() will keep the source file comment and protection bits (except A,R,D because BBS files better have R and D set)

RESULT

Destination - The passed FAMEUlPathList structure with RESULT pointing to the list element which contains the name of the directory where the destination file was posted to. So you may easily check this out.

RESULT is NULL if an AmigaDOS Error occured, if there was no directory with enough free space left (ERROR_DISK_FULL), or if the destination file already exists in one of the given directories (ERROR_OBJECT_EXISTS). Whatever happens: if RESULT = NULL, use IoErr() to get the fault.

If FPFF_CHECKONLY was set, RESULT will be the list element containing the path the file was found, or NULL if the file doesn't exist anywhere.

```
                        With the CHECKONLY flag, RESULT does not say
                        anything about if there is any path with enough
                        free space to store the file. Don't mischange..

NOTES
      You may combine FPFF_REPLACE and FPFF_NODELETE, but you shouldn't
      ever combine FPFF_CHECKONLY with FPFF_REPLACE or FPFF_NODELETE.

      Beyond AmigaDOS calls, this function uses 1400 bytes of stack space.
```

1.30 FAME.library/FAMEResetPool

```
               NAME
      FAMEResetPool -- reset a pool and free all it's childs

SYNOPSIS
      FAMEResetPool(FAMEMemPool)
                    A1

      VOID FAMEResetPool(APTR)

FUNCTION
      This function frees all child pools and resets the main pool.
      Useful to free all allocated memory areas (including those which
      have been part of the main pool) with one single call while keeping
      the main pool available for further allocations. This saves an addi-
      tional FAMECreatePool() call and reduces memory fragmentation.

INPUTS
      FAMEMemPool     - the pool you want to reset

SEE ALSO

              FAMEAllocPooled()
              ,
              FAMECreatePool()
              ,
              FAMEDeletePool()
              ,
              FAMEFreePooled()
```

1.31 FAME.library/FAMEReverseLong

```
               NAME
      FAMEReverseLong -- reverse byte order of a long sized value

SYNOPSIS
      FAMEReverseLong(Value)
                      d0

      ULONG FAMEReverseLong(ULONG Value)
```

```
FUNCTION
      Reverses the low/high byte order of the given Value.
      The two reverse functions have been written for non-asm programmers
      who must handle PC data structures containing WORD or LONG data
      (may be also pointers). 80x86 CPUs (and some other CPUs, like the
      C64's 6510) store data in reversed order. A 680x0 value of $12345678
      equals $78563412 in 80x86 format.

INPUTS
      Value           - The value to convert

RESULT
      NewValue        - The converted (reversed) value

NOTES
      Except d0 (Result), this function is guaranteed to preserve all
      registers.
      Because Inputs and Result are defined as ULONG, C language programmers
      may need type casting with e.g. APTR type values.

SEE ALSO

               FAMEReverseWord()
```

1.32 FAME.library/FAMEReverseWord

```
               NAME
      FAMEReverseWord -- reverse byte order of a word sized value

SYNOPSIS
      FAMEReverseWord(Value)
                      d0

      UWORD FAMEReverseWord(UWORD Value)

FUNCTION
      Reverses the low/high byte order of the given Value.
      The two reverse functions have been written for non-asm programmers
      who must handle PC data structures containing WORD or LONG data
      (may be also pointers). 80x86 CPUs (and some other CPUs, like the
      C64's 6510) store data in reversed order. A 680x0 value of $1234
      equals $3412 in 80x86 format.

INPUTS
      Value           - The value to convert

RESULT
      NewValue        - The converted (reversed) value

NOTES
      Except d0 (Result), this function is guaranteed to preserve all
      registers.
      Because Inputs and Result are defined as UWORD, C language programmers
      may need type casting with, for example, SHORT type values.
```

```
SEE ALSO

               FAMEReverseLong()
```

1.33 FAME.library/FAMEStackReport

```
NAME
      FAMEStackReport -- Retrieve the remaining stack space of your program

SYNOPSIS
      AvailStack = FAMEStackReport()
      D0

      LONG FAMEStackReport(VOID)

FUNCTION
      Tells you about how many bytes are left on your stack. This function
      is used for debugging, and it can be used to make a program more
      safe against stack overflowing. But note that permanent stack checks
      may slow down your program.

RESULT
      AvailStack - The remaining stack size.
```

1.34 FAME.library/FAMEStartECTimer

```
                NAME
      FAMEStartECTimer -- Start an E-Clock time measure

SYNOPSIS
      FAMEStartECTimer(Dest)
                       A0

      VOID FAMEStartECTimer(struct timeval *dest)

FUNCTION
      This function fills a timeval structure with the current E-Clock
      time. Together with
              FAMEStopECTimer()
              , you may perform easy time
      measuring.
      After calling this function, you may call
              FAMEStopECTimer()
              ,
      passing the same timeval structure, where the time difference between
      the old structure contents and the new E-Clock time gets written back.

INPUTS
      Dest      - A timeval structure. This structure gets filled
                  with the current E-Clock values.

NOTES
```

```
       There is *no* need to have any calls to
               FAMEStopECTimer()
               . Also,
       you don't need to free anything. You may call this function multiple
       times using different timeval structures.

       This function preserves A0.

SEE ALSO

                FAMEStopECTimer()
```

1.35 FAME.library/FAMEStopECTimer

```
                NAME
       FAMEStopECTimer -- Stop an E-Clock time measure

SYNOPSIS
       FAMEStopECTimer(Dest)
                       A0

       VOID FAMEStopECTimer(struct timeval *dest)

FUNCTION
       This function stops a
               FAMEStartECTimer()
                time measuring operation.
       To be exact, this function does nothing more than reading the current
       E-Clock again and then subtracting the result from the passed timeval
       structure.

INPUTS
       Dest            - A timeval structure (the same structure you already
                         passed to
               FAMEStartECTimer()
               .
                         This structure gets filled with the time difference
                         between both FAMEStartECTimer() and FAMEStopECTimer()
                         calls.
NOTE
       This function preserves A0 and A1, while D0 (Lo) and D1 (Hi) contain
       the results of the second E-Clock measure (not the time difference).

SEE ALSO

                FAMEStartECTimer()
```

1.36 FAME.library/FAMEStrChr

```
                NAME
       FAMEStrChr -- Find a character in a string
```

```
SYNOPSIS
      NewString = FAMEStrChr(Source, MatchChar)
      D0                      A0        D0

      STRPTR FAMEStrChr(STRPTR Source, UBYTE MatchChar)

FUNCTION
      Find a character in a string.
      If a match is found, NewString will point to the first match inside
      the string. If no match was found, RESULT will be NULL.

INPUTS
      Source    - The source string.
      MatchChar - The char to search for.

RESULT
      NewString - The updated string pointer.

SEE ALSO

               FAMEStrChrCase()
              ,
               FAMEStrStr()
              ,
               FAMEStrStrCase()
```

1.37 FAME.library/FAMEStrChrCase

```
                NAME
      FAMEStrChrCase -- Find a character in a string

SYNOPSIS
      NewString = FAMEStrChrCase(Source, MatchChar)
      D0                           A0        D0

      STRPTR FAMEStrChrCase(STRPTR Source, UBYTE MatchChar)

FUNCTION
      Find a character in a string.
      If a match is found, NewString will point to the first match inside
      the string. If no match was found, RESULT will be NULL.

INPUTS
      Source    - The source string.
      MatchChar - The char to search for.

RESULT
      NewString - The updated string pointer.

SEE ALSO

               FAMEStrChr()
              ,
               FAMEStrStr()
              ,
```

```
                FAMEStrStrCase()
```

1.38 FAME.library/FAMEStrCopy

```
NAME
      FAMEStrCopy -- Copy a string to another until MaxLen has been reached


SYNOPSIS
      NewString = FAMEStrCopy(Source, Destination, MaxLen)
      D0                      A0          A1          D0

      STRPTR FAMEStrCopy(STRPTR Source, STRPTR Destination, UWORD MaxLen)

FUNCTION
      Copy a string into another with a limit of MaxLen.
      FAMEStrCopy stops if a NULL byte was found, or if MaxLen is
      reached. In this case, a NULL byte gets added, so remember that
      the size of your destination buffer must be at least MaxLen+1 bytes.

INPUTS
      Source      - The source string.
      Destination - The destination string.
      MaxLen      - Maximum string copy size.

RESULT
      NewString   - The same as Destination. Ask some C coders why. :^)
```

1.39 FAME.library/FAMEStrCut

```
                NAME
      FAMEStrCut -- cut a string from a search string position

SYNOPSIS
      FAMEStrCut(String, CutString, MaxSearchRange)
                 A0      A1         D0

      STRPTR FAMEStrCut(STRPTR, STRPTR, ULONG)

FUNCTION
      Cut a string from the first position of a part string.
      The part string will be searched inside the source string; if
      a match was found, the right part of the source string gets cut
      at the start position of the search string. If nothing was found,
      nothing gets changed. This function is not case-sensitive;
      german umlauts and other international chars are not considered.

INPUTS
      String          - The string to truncate
      CutString       - The string part to search for. The length of the
                        CutString is limited to 512-1 bytes.
      MaxSearchRange  - Important -the maximum string length to search for
```

```
                        the CutString. Normally, this is SizeOf(String).

RESULT (V2)
      Result points to the (new) end position of the passed string, no
      matter if the CutString was found or not. Result will point exactly
      to the trailing NULL byte of the string.

NOTE
      This function uses 528 Bytes of Stack space.

SEE ALSO

               FAMEStrCutCase()
              ,
               FAMEChrCut()
              ,
               FAMEChrCutCase()
```

1.40 FAME.library/FAMEStrCutCase

```
                NAME
      FAMEStrCutCase -- cut a string from a search string position

SYNOPSIS
      FAMEStrCutCase(String, CutString, MaxSearchRange)
                      A0       A1            D0

      STRPTR FAMEStrCutCase(STRPTR, STRPTR, ULONG)

FUNCTION
      Cut a string from the first position of a part string.
      The part string will be searched inside the source string; if
      a match was found, the right part of the source string gets cut
      at the start position of the search string. If nothing was found,
      nothing gets changed. This function is equal to
              FAMEStrCut()
              ,
      but case-sensitive and also faster.

INPUTS
      String          - The string to truncate
      CutString       - The string part to search for
      MaxSearchRange  - Important -the maximum string length to search for
                        the CutString. Normally, this is SizeOf(String).

RESULT (V2)
      Result points to the (new) end position of the passed string, no
      matter if the CutString was found or not. Result will point exactly
      to the trailing NULL byte of the string.

SEE ALSO

               FAMEStrCut()
              ,
               FAMEChrCut()
```

```
,
 FAMEChrCutCase()
```

1.41 FAME.library/FAMEStrFil

```
                NAME
      FAMEStrFil -- Fill string buffer with a char

SYNOPSIS
      FAMEStrFil(FillBuffer, FillChar, NumberOfChars)
                      A0        D0           D1

      STRPTR FAMEStrFil(STRPTR FillBuffer, UBYTE FillChar, WORD NumberOfChars)

FUNCTION
      Fill up the buffer with a char up to NumberOfChars. After that,
      a NULL byte gets added to close the string.

INPUTS
      FillBuffer    - The buffer to fill up.
      FillChar      - The fill character.
      NumberOfChars - The number of characters to write (WORD size !).

RESULT (V2)
      Points to the trailing NULL byte at the end of the String.

SEE ALSO

               FAMEFillMem()
              ,
               FAMEMemSet()
```

1.42 FAME.library/FAMEStrMid

```
NAME
      FAMEStrMid -- Return a substring from a string

SYNOPSIS
      Result = FAMEStrMid(Source, Destination, StartPos, NumberOfChars)
      D0                    A0          A1          D1           D0

      LONG FAMEStrMid(STRPTR Source, STRPTR Destination, LONG StartPos, LONG  ↩
         NumberOfChars)

FUNCTION
      FAMEStrMid() writes a string part of the source string to the destina-
      tion string, beginning at character position StartPos, and with a
      length of NumberOfChars. If NumberOfChars is higher than the remaining
      bytes of the source string, the rest of the string is copied to the
      the destination string, not more. If you want to copy the right part
      of a string, you may specify -1 for NumberOfChars.
```

```
      The destination string gets null-terminated.

INPUTS
      Source        - The source string.
      Destination   - The destination string.
      StartPos      - Start position for copying.
      NumberOfChars - Number of chars to copy.

RESULT
      Result        - -1 if StartPos is beyond Source, else NULL.
```

1.43 FAME.library/FAMEStrStr

```
                NAME
      FAMEStrStr -- Find a string in a string


SYNOPSIS
      NewString = FAMEStrStr(Source, MatchString)
      D0                      A0         A1

      STRPTR FAMEStrStr(STRPTR Source, STRPTR MatchString)

FUNCTION
      Find a string in a string.
      If a match is found, NewString will be the updated string pointer
      pointing to the first match. If no match can be found, NewString
      will be NULL.

INPUTS
      Source      - The source string.
      Matchstring - The string which FAMEStrStr is been searching for.

RESULT
      NewString   - A pointer to the string position of the search string,
                    or NULL if nothing was found.

NOTE
      Both strings don't get modified anyway.

SEE ALSO

               FAMEStrStrCase()
              ,
               FAMEStrChr()
              ,
               FAMEStrChrCase()
```

1.44 FAME.library/FAMEStrStrCase

```
                NAME
      FAMEStrStrCase -- Find a string in a string
```

```
SYNOPSIS
      NewString = FAMEStrStrCase(Source, MatchString)
      D0                         A0         A1

      STRPTR FAMEStrStrCase(STRPTR Source, STRPTR MatchString)

FUNCTION
      Find a string in a string. If a match is found, NewString will be
      the updated string pointer to the first match. If no match can be
      found, NewString will be NULL.

INPUTS
      Source      - The source string.
      Matchstring - The string which FAMEStrStrCase is been searching for.

RESULT
      NewString   - A pointer to the string position of the search string,
                    or NULL if nothing was found.

SEE ALSO

               FAMEStrStr()
              ,
               FAMEStrChr()
              ,
               FAMEStrChrCase()
```

1.45 FAME.library/FAMEStrToLower

```
                NAME
      FAMEStrToLower -- lower-case a string


SYNOPSIS
      String = FAMEStrToLower(String)
      D0                      A0

      STRPTR FAMEStrToLower(STRPTR String)

FUNCTION
    Lower-case a string. Result is the same string you passed.
      German umlauts and other international chars are not considered.

INPUTS
      String  - The string to convert.

RESULT
      String  - A pointer to the converted string (same as input).

SEE ALSO

               FAMEStrToUpper()
```

1.46 FAME.library/FAMEStrToUpper

```
                NAME
      FAMEStrToUpper -- upper-case a string


SYNOPSIS
      String = FAMEStrToUpper(String)
      D0                      A0

      STRPTR FAMEStrToUpper(STRPTR String)

FUNCTION
      Upper-case a string. Result is the same string you passed.
      German umlauts and other international chars are not considered.

INPUTS
      String  - The string to convert.

RESULT
      String  - A pointer to the converted string (same as input).

SEE ALSO

               FAMEStrToLower()
```

1.47 FAME.library/FAMESub64

```
                NAME
      FAMESub64       -- perform a QuadWord (64-bit) subtraction

SYNOPSIS
      FAMESub64(SrcHi, SrcLo, Destination)
                 d0     d1        a0

      VOID FAMESub64(ULONG, ULONG, APTR)

FUNCTION
      Subtracts an unsigned 8-byte value (SrcLo/SrcHi) from a QuadWord
      at a specified destination address.

INPUTS
      SrcHi           - The upper LONG of the 64-bit value to subtract.
                        If you want to subtract a LONG from the destination
                        value (instead of a QuadWord), set SrcHi to NULL
                        and pass the LONG to be subtracted in SrcLo.

      SrcLo           - The lower LONG of the 64-bit value to subtract.

      Destination     - A pointer to a memory address of 8 bytes
                        in size where SrcHi and SrcLo get subtracted from.
                        These 8 bytes build one QuadWord. If the first
                        half of this area (the first LONG) is NULL,
                        the QuadWord value equals the lower LONG
```

```
                        value and will be inside a normal range from
                        0 to 4,294,967,295.
                        If the upper LONG inon-zero, the 64-bit value
                        (Destination) equals:
                        (UpperLong * 4,294,967,296) + LowerLong.

EXAMPLE
      Destination may be a structure, or a part of a structure:

              struct  MyData {
              ULONG   ULBytesHi;
              ULONG   ULBytesLo;
              };

      The function call may look lik "FAe this:

      FAMEAdd64(NULL, BytesForThisUpload, MyData)

NOTE
      The library doesn't check for overflows. You may also use this function
      for calculating signed values, but then you'll have to interpret the
      destination value yourself. For example, if you subtract 1 from a
      value of 0, then the result will be $FFFFFFFF FFFFFFFF.

SEE ALSO

               FAMEAdd64()
```

1.48 FAME.library/FAMESwapRedWhite

```
NAME
      FAMESwapRedWhite -- Swap red and white ANSI color codes in a string

SYNOPSIS
      FAMESwapRedWhite(String)
                        A0

      STRPTR FAMESwapRedWhite(STRPTR)

FUNCTION
      Swap all ANSI red/white colour commands in a string. The passed string
      gets converted. No Result.
      Normally, this function is never used by FIM coders, because FAME
      does this itself on every ANSI string, depending on the user's color
      setup. FIM coders always use ESC[31m for WHITE foreground colour,
      and ESC[37m for red colour (even if real ANSI uses 31 for red, and
      37 for white). We have decided to use 31 (or 41) as white and 37/47
      as red colour for FIM doors.

INPUTS
      String      - The string to convert.

RESULT (V2)
```

```
Points to the passed string (same as input).
```

1.49 FAME_library.Guide: Contents

```
                              FAME.library V 4.00 function reference
                ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

               Introduction

               Debug functions

               File operations

               Memory functions

               Miscellaneous things

               String operations
                    -> Complete function overview

               Notes
                      ---------------------------------------------
                      All Functions
          ------------------------------------------------


               FAMEAdd64()

               FAMEAllocObject()

               FAMEAllocPooled()

               FAMEAvailExe()

               FAMEChrCut()

               FAMEChrCutCase()

               FAMECreatePool()

               FAMECutANSI()

               FAMEDeletePool()

               FAMEDosMove()

               FAMEExecuteDir()

               FAMEFileCopy()

               FAMEFillMem()

               FAMEFreeDevInfoList()

               FAMEFreeDiskSpace()
```

FAMEFreeExecuteDirList()

FAMEFreeFile()

FAMEFreeObject()

FAMEFreePooled()

FAMEFSearch()

FAMEGetDevInfoList()

FAMEIsNumStr()

FAMELoadFile()

FAMELoadFilePooled()

FAMEMemSet()

FAMENum64ToStr()

FAMENumToStr()

FAMEOverallBytes()

FAMEPostFile()

FAMEResetPool()

FAMEReverseLong()

FAMEReverseWord()

FAMEStackReport()

FAMEStartECTimer()

FAMEStopECTimer()

FAMEStrChr()

FAMEStrChrCase()

FAMEStrCopy()

FAMEStrCut()

FAMEStrCutCase()

FAMEStrFil()

FAMEStrMid()

FAMEStrStr()

```
            FAMEStrStrCase()

            FAMEStrToLower()

            FAMEStrToUpper()

            FAMESub64()

            FAMESwapRedWhite()
```

1.50 FAME_library.Guide: Debug Functions

```
                              FAME.library V 4.00 function reference
              ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

            Introduction
                 -> Debug functions

            File operations

            Memory functions

            Miscellaneous things

            String operations

            Complete function list

            Notes
                   ---------------------------------------------
                   Debug Functions
        -------------------------------------------


            FAMEStackReport()
```

1.51 FAME_library.Guide: File Operations

```
                              FAME.library V 4.00 function reference
              ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

            Introduction

            Debug functions
                 -> File operations

            Memory functions

            Miscellaneous things

            String operations
```

1.52 FAME_library.Guide: Memory Functions

FAME.library V 4.00 function reference
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

```
            FAMEResetPool()
```

1.53 FAME_library.Guide: Miscellaneous Things

```
                              FAME.library V 4.00 function reference
              ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

            Introduction

            Debug functions

            File operations

            Memory functions
                 -> Miscellaneous things

            String operations

            Complete function overview

            Notes
                   ---------------------------------------------
                   Miscellaneous Things
    ------------------------------------------------


            FAMEAdd64()

            FAMEAllocObject()

            FAMEAvailExe()

            FAMEExecuteDir()

            FAMEFreeDevInfoList()

            FAMEFreeDiskSpace()

            FAMEFreeExecuteDirList()

            FAMEFreeObject()

            FAMEGetDevInfoList()

            FAMEOverallBytes()

            FAMEPostFile()

            FAMEReverseLong()

            FAMEReverseWord()

            FAMEStartECTimer()
```

FAMEStopECTimer()

FAMESub64()

1.54 FAME_library.Guide: String Operations

```
                        FAME.library V 4.00 function reference
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Introduction

Debug functions

File operations

Memory functions

Miscellaneous things
     -> String operations


Complete function overview

Notes
       ------------------------------------------
       String Operations
-----------------------------------------------


FAMEChrCut()

FAMEChrCutCase()

FAMECutANSI()

FAMEIsNumStr()

FAMENum64ToStr()

FAMENumToStr()

FAMEStrChr()

FAMEStrChrCase()

FAMEStrCopy()

FAMEStrCut()

FAMEStrCutCase()

FAMEStrFil()

FAMEStrMid()
```

1.55 FAME.library/Notes

```
                        FAME.library V 4.00 function reference
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

          Introduction

          Debug functions

          File operations

          Memory functions

          Miscellaneous things

          String operations

          Complete function overview
               -> Notes

-----------------------------------------------
                    Notes
-----------------------------------------------

Locale notes:

Some FAME.library functions support Locale features, but all
functions never fail if any locale.library problem occurs.
Locale.library version must be V38 or higher.


General notes:

FAME.library needs AmigaOS V37 (Kickstart 2.05) or higher,
otherwise any attempt to open FAME.library will fail.

As long as not especially stated, most FAME.library functions
need less than 100 bytes of stack space; some of the functions
call AmigaOS functions which need some more. However, a minumum
stack size of 4096 bytes is suggested by Commodore and should
normally be enough.

Some FAME library functions are very simple, thought for non-
asm programmers. For maximum speed, asm writers may use their
own inline code. I think that asm developers will know well
```

about which of the functions must be no doubt extremely small.

1.56 FAME.library/Introduction

Welcome to FAME.library !

FAME.library is a public shared library, first developed for having some
fast assembler code for use by the FAME BBS only, but after some time,
many functions have been built in which may be useful for everybody;
not only for FAME BBS or FAME door programmmers.

So, here it is. FAME.library is a public shared function library, which may
be freely spread everywhere, as long as both library and this document always
get spread together, and in unchanged state.

FAME.library is SHAREWARE. Developers who want to include FAME.library
in PD or FREEWARE programs, wre absolutely *no profit* is gained anyway,
may use FAME.library for FREE. Also, all users of *any kinda program* which
makes use of FAME.library, do not have to pay anything for FAME.library.
The ShareWare donation must only be "paid" by developers who want to use
FAME.library within some commercial or ShareWare program(s): the ShareWare
"amount" for FAME.library is one free and fully registered copy of your
program(s), registered to me, the author of FAME.library. Thats all.
So this kind of ShareWare donation "costs" you not a single buck. :^)

Note: FAME.library contains some special FAME BBS functions, only used
by the FAME BBS and their doors, so that other programs, which have nothing
to do with BBS systems, don't have any use of, but this handsome of functions
really doesn't consume much memory. Some of these functions are declared
as private, because i don't think that any other program could have any use
of these things. I suggest *not* to try using FAME private functions, because
everything may change here with future updates.

What is FAME.library good for ?

Good question, simple answer: FAME.library may be good for anything which
came in my mind which was worth to be a public shared library function.

Many functions replace ANSI C code (c programmers may reduce code size by
replacing many ANSI C functions by FAME.library calls), some functions are
FAME BBS functions, mostly used by FAME and it's door programs, and the
remaining ones are miscellaneous things; anything i thought about "how
nice if this was a library function". Just have a look into the function
overview.

Contact Address

E-Mail: Bloodrock@funboard.in-berlin.de

Attention: my E-Mail address is not valid at the moment.
Try to contact me by writing mail to my account at the Punishment Inc. BBS:

+49 30 694-8470 / 694-8570.

About

FAME.library is © by Oliver "Bloodrock" Lange.

FAME BBS is © by David "Strider" Wettig.

Amiga is a registered Trademark (of some company. Who knows which ?)