MECAFF

# Multiline External Console And Fullscreen Facility

for VM/370 R6 SixPack 1.2

***An informal guide to some internals of MECAFF Version 1.0.0***

Dr. Hans-Walter Latz, 2012

## 1 General

Only few software documentations are *really* comprehensive and complete, and this one does not even attempt to.

So if some information is missing here or in the MECAFF manual, the following more or less traditional approaches may be useful for understanding the internals or when hunting bugs:

- Do not wait for a sonorous voice in your head saying:

  *Use the SOURCE, Luke*

- Play with the beast, after all, the thing is virtual in many senses, so setting up a new Hercules mainframe with a VM/370 to prevent important things (your own work) to get broken is not really a problem (unless disk space is tight)
- Ask somebody who might know, for example on the H390-VM group (where the source and binaries for MECAFF are available)

## 2 Content of the source archive

The ZIP mecaff-src-1.0.0-zip contains the following files:

| | |
|---|---|
| `mecaff-cms-src.aws` | the sources for the CMS part of MECAFF |
| `mecaff-help.aws` | the CMS help files for MECAFF |
| `mecaff-java-src.zip` | the sources for the Java-part of MECAFF |
| `MECAFF-Manual-1.0.0.docx` | the MS-Word document for the MECAFF manual |
| `MECAFF-Internals-1.0.0.docx` | the MS-Word document for the MECAFF internals guide |
| `MECAFF-Internals-1.0.0.pdf` | the PDF-version MECAFF internals guide (this file) |

# 3 Communication protocol

## 3.1 Overview

The requests sent from FSIO to the MECAFF process are embedded in the output lines sent from the VM through the standard console output.

A request is introduced at the line start by the character sequence

```
<{>}
```

followed by a *request identifier* character identifying the request and – depending on the request type – encoded parameters. Depending on the connection style (CONS or GRAF) the complete command fits on a single output line written via WRTERM (max. 130 characters) if CONS-style or on 21 x 80 characters written via DMSGIO if GRAF-style. Longer data sequences are sent with multiple consecutive requests, with the last request completing the data being identified as final by the request identifier.
However, the request GET-TERM-DATA is always written via WRTERM, as the connection style is unknown at this point.

If a request has a response, FSIO waits for the response to come in after sending the request.

The responses from the MECAFF-process are also introduced by the character sequence

```
<{>}
```

followed by a *response identifier* character identifying the response type and optional encoded data. The responses are always sent as input to the standard VM-prompt and are thereof limited to 130 characters, so longer data sequences are sent as consecutive responses, with the last response making up the complete data identified as final response by the response code. FSIO actively requests the transmission of a response by doing RDTERM, which triggers the MECAFF-process to send the (next) response.

Encoding of data differentiates between integers and arbitrary data text. The encoding uses a character mapping with changing character subsets along the encoded sequence to make it easier to recognize lost characters or characters streams not belonging to the FSIO/MECAFF-protocol. Some requests and responses have string parameters as plain text (not encoded).

Unknown requests or requests where encoding errors are recognized will not be accepted as commands (i.e. will not be silently ignored), instead the complete text starting with the introducing character sequence will be treated as normal output from the VM and displayed on the console.

## 3.2 Integer-Encoding

Integers are encoded from the raw 4 bytes, starting with "upper" (highest significant) 4-bit nibbles, skipping leading nibbles with value zero, using the following scheme:

- Non-last nibbles are translated into one of the characters `abcdefghjklmnopq`, using the nibble value as index.
- The last (lower 4 bits) nibble is translated into one of the characters `ABCDEFGHJKLMNOPQ`, using the nibble value as index.

## 3.3 Data -Encoding

Arbitrary byte sequences (data or EBCDIC strings) are encoded with 2 characters per byte, with different encoding schemes for the non-last and the last byte:

- Non-last bytes:
    - The upper 4-bit nibble is translated into one of the characters `ABCDEFGHJKLMNOPQ`, using the nibble value as index.
    - The lower 4-bit nibble is translated into one of the characters `STUVWXYZ23456789`, using the nibble value as index.
- Last Byte
    - The upper 4-bit nibble is translated into one of the characters `bcdefghiklmnopqr`, using the nibble value as index.
    - The lower 4-bit nibble is translated into one of the characters `ABCDEFGHJKLMNOPQ`, using the nibble value as index.

## 3.4 Get terminal/console properties

The following requests/responses are used by the FSIO routines `__qtrm()`, `__qtrm2()`, `__qtrmpf()` and `__fsqvrs()`.

### 3.4.1 Request GET-TERM-DATA

Verify that the terminal is connected through a MECAFF-process and get the properties of the terminal and the MECAFF-console.

Request identifier: T

Parameters:

- Plain text requesting the user to simply press enter if the request is displayed on the screen (i.e. the terminal is not attached to a MECAFF-process)

### 3.4.2 Response GET-TERM-DATA

Response to the GET-TERM-DATA request.

Response identifier: T

Response data:

- Integer: *transportVersion*
  identification of the historical version of the response and supported features, current: 3
- Data: *terminalType*
  as identified during the TN3270 negotiation, e.g. IBM-3278-4-E
- Integer: *numAltRows*
  number of lines in the alternate screen
- Integer: *numAltCols*
  number of columns in the alternate screen
- Integer: *canAltScreenSize*
  1 if an alternate screen is supported/available, 0 if not
- Integer: *canExtHighlight*
  1 if extended highlighting is supported by the terminal, 0 if not

- Integer: *canColors*
  1 if it is a color terminal, 0 if not
- Integer: *sessionId*
  used to recognize if the terminal was disconnected and reconnected while the fullscreen application communicates with MECAFF
- Integer: *sessionMode*
  3270 for GRAF-style and 3215 for CONS-style
- Screen attributes: 5 pairs of integers for the {*element-type*, *attribute*}-tuples, with the first integer of the tuple encoding the index of the screen element type, the second integer encoding the color index (+100 if highlighted)
- Assigned PF-keys: one integer having the bits 1..24 set to 1 if the corresponding PF-key (indexed by the bit position) has a command assigned.
- Integer: *majorMecaffVersion*
  Major-part of the version of the MECAFF-version
- Integer: *minorMecaffVersion*
  Minor-part of the version of the MECAFF-version
- Integer: *subMecaffVersion*
  Subversion-part of the version of the MECAFF-version

#### 3.4.3 Request GET-TERM-PFDATA

Get the command assigned to a PF-key.

Request identifier: `t`

Request data:

- Integer: *pfNo*
  the PF-key queried.

#### 3.4.4 Response GET-TERM-PFDATA

Response to the GET-TERM-PFDATA request for a particular *pfNo*.

Response identifier: `t`

Response data:

- Integer: *commandLength*
  length of the command text associated to the PF-key, with 0 if no command is assigned.
- (optional) Plain text: *commandText*
  the command text for the PF-key, with this response field missing if the *commandLength* is 0

### 3.5 Setting console properties

The following requests/responses are used by the FSIO routines `_strmat()` and `__strmpf()`.

#### 3.5.1 Request SET-CONSOLE-CFG

Set a configuration parameter for the MECAFF-console, either the display attribute for screen element types or a PF-key command assignment.

Request identifier: `C`

Request data:

- Integer: *cfgItem*
  identification of the item to be set, with the values:
  1..100: number of screen element attribute encodings following
  101..124: PF-key to assigned.
- If *cfgItem* < 100: sequence on Integer-pairs for the {*element-type, attribute*}-tuples to set (see the GET-TERM-DATA response)
- If *cfgItem* > 100: plain text: `[`*commandText*`]`
  the command text to be assigned to PF key, with the square brackets transmitted but not belonging to the command; setting an empty command text clears the command currently assigned to the key.

### 3.5.2 Response SET-CONSOLE-CFG

Response to the SET-CONSOLE-CFG request.

Response identifier: `C`

Response data:

- Integer: *returncode*
  currently always 0 (success), as errors are ignored.

## 3.6 Fullscreen write

The following requests/responses are used by the FSIO routine `__fswr()`. Doing a fullscreen write involves the following steps:

- Acquire the screen with INIT-FULLSCREEN-MODE, passing the command type of the 3270 output stream as indication if the previous fullscreen state must still be displayed (i.e. the CCW is Write or EraseAllUnprotected, which will only work as intended if the MECAFF-console did not take back the ownership on the screen).
- If the response to INIT-FULLSCREEN-MODE indicates that the screen is now owned by the program using FSIO:
  - split the 3270 output buffer in *n* chunks sized for the connection mode encoded as *Data*
  - send the first to (n-1)-nth chunks with a WRITE-FULLSCREEN-CHUNK request
  - send the n-nth (last) chunk with a WRITE-FULLSCREEN-CHUNK-FINAL request

### 3.6.1 Request INIT-FULLSCREEN-MODE

Request the ownership on the screen.

Request identifier: `W`

Request data:

- Integer: *sessionId*
  the *sessionId* for this connection, retrieved with the initial `__fsqry()`.
- Integer: *commandType*
  the type of (a supported) CCW command for the 3270 output stream to be sent, with:
  1 = Write-CCW

2 = EraseWrite-CCW
3 = EraseWriteAlternate-CCW
4 = EraseAllUnprotected-CCW

- Integer: *rawDataLength*
  the total length of the 3270 output buffer which will be sent.

### 3.6.2 Response INIT-FULLSCREEN-MODE

Response to the INIT-FULLSCREEN-MODE request.

Response identifier: `W`

Response data:

- Integer: *returncode*
  result of the screen ownership acquisition attempt:
  0 : success, the program now owns the screen and may send the 3270 output stream
  1 : failed, CCW requires the old fullscreen content, but this was lost (or never written)
  2 : failed, wrong session id.

### 3.6.3 Request WRITE-FULLSCREEN-CHUNK

Send the next non-final chunk of the 3270 output stream, with the new fragment being appended to the previous ones in an internal buffer.

Request identifier: `f`

Request data:

- Data: *fragment*
  the (next) fragment of the encoded data stream.

### 3.6.4 Request WRITE-FULLSCREEN-CHUNK-FINAL

Send the last chunk of the 3270 output stream, initiating the actual transmission of the collected data stream to the terminal.

Request identifier: `F`

Request data:

- Data: *fragment*
  the last fragment of the encoded data stream.

## 3.7 Fullscreen read

The following requests/responses are used by the FSIO routines `__fsrdp()` and `__fsrd()`. Doing a fullscreen read involves the following steps:

- Start fullscreen read operation with READ-FULLSCREEN-INPUT, with several variants like timed out read or polling resp. querying fullscreen input state.
- Depending on the read operation or the current fullscreen (output/input) state:
  - a response is returned immediately with the error or the queried fullscreen read state

- the MECAFF-process waits for a 3270 input stream sent by the terminal
  - if waiting times out, a response with fullscreen read state (“timed out”) is sent
- At the end of a successful fullscreen read operation , the 3270 input stream is splitted in *n* fragments, each fragment is encoded as *Data* and sent to the host:
  - the first (n-1)-nth chunks are sent with a READ-FULLSCREEN-CHUNK
  - the (last) n-nth chunk is sent with a READ-FULLSCREEN-CHUNK-FINAL

### 3.7.1 Request READ-FULLSCREEN-INPUT

Request to start a fullscreen read interaction or management operation.

Request identifier: `I`

Request data:

- Integer: *sessionId*
  the *sessionId* for this connection, retrieved with the initial `__fsqry()`.
- Integer: *timeout*
  the fullscreen read timeout interval, expressed in 1/10 seconds, with the following special values:
  `-42424242`: stop waiting asynchronously for user input (cancel current fullscreen read)
  `< 0`: query user input availability status
  `0`: query user input availability status and send input data if available
  `> 0`: wait for user's fullscreen input with the given timeout value.
- Integer: *gracePeriod*
  time interval in 1/10 seconds to keep the screen ownership with the fullscreen program if the read timeout occurred: the program loses ownership only if the next fullscreen operation arrives after the grace period.

### 3.7.2 Response READ-FULLSCREEN-INPUT

Response to the READ-FULLSCREEN-INPUT request in case of an error, of a timeout or when polling for fullscreen state.

Response identifier: `E`

Response data:

- Integer: *returncode*
  with the following value meanings:
  0 : fullscreen input is available (as response to a query user input availability status)
  1 : the last (full-)screen was overwritten due to asynchronous output form VM, the screen must be re-written before doing the fullscreen read.
  2 : wrong session id, request rejected.
  3 : the last fullscreen read operation timed out
  4 : no fullscreen input is available (as response to a query user input availability status)

### 3.7.3 Response READ-FULLSCREEN-CHUNK

Transfer the next non-last fragment of the current 3270 input stream to the host.

Response identifier: `i`

Response data:

- Data: *inputStreamFragment*
  the next fragment of the complete 3270 input stream.

#### 3.7.4 Response READ-FULLSCREEN-CHUNK-FINAL

Transfer the last fragment of the current Data-encoded 3270 input stream to the host, signaling that the transmission ends and processing of the input stream can begin.

Response identifier: `I`

Response data:

- Data: *inputStreamFragment*
  the last fragment of the complete 3270 input stream.

## 3.8 Misc

### 3.8.1 Request SET-FULLSCREENLOCK-TIMEOUT

Set the timeout value that will delay losing the screen ownership by the fullscreen program. This request is used by the FSIO-routine `__fslkto()`.

Request identifier: &

Parameters:

- Integer: *lockinTimeoutValue*
  specific timeout in 1/10 seconds to be used for delay losing the screen ownership, with the value 0 resetting this timeout value.

### 3.8.2 Request ECHO

Send back the parameter of the request as command input to the host as if the user had entered it at the input prompt of the MECAFF-console.

Request identifier: `:`

Parameters:

- Plain text: *stringToBeEchoed*
  text (not encoded) to be sent back as command input.

## 4 Sources on the CMS side

**Warning**: avoid loading the files from the MECAFF sources tape to the A minidisk, as this will probably overwrite your `PROFILE EE` file.

### 4.1 Content of the sources tape

```
 IS-STALE C        A1
 ASM      EXEC     A1
 CC       EXEC     A1
 CMSCOMP  EXEC     A1
 FSCOMP   EXEC     A1
 FSLINK   EXEC     A1
 GLBLPRE  H        A1
 GLBLPOST H        A1
 BOOL     H        A1
 FSIO     H        A1
 CMSMIN   ASSEMBLE A1
 WR3270   ASSEMBLE A1
 FSIO     C        A1
 FS-QRY   C        A1
 FS-CTL   C        A1
 FS3270   H        A1
 AID3270  H        A1
 FS3270   C        A1
 EEUTIL   H        A1
 EEUTL1   C        A1
 EEUTL2   C        A1
 EECORE   H        A1
 EESCRN   H        A1
 EECORE   C        A1
 EESCRN   C        A1
 EEMAIN   H        A1
 EEMAIN   C        A1
 EECMDS   C        A1
 EEPREFIX C        A1
 EELIST   C        A1
 EEHELP   C        A1
 FSHELP   C        A1
 IND$FILE H        A1
 IND$DENC C        A1
 IND$SCRN C        A1
 IND$FILE C        A1
 EBCDIC   MEMO     A1
 VISTA    IND$MAP  A1
 FSRDTEST C        A1
 SYSPROF  EE       A1
 PROFILE  EE       A1
 CKMECAFF EXEC     A1
 FSLIST   EXEC     A1
 FSVIEW   EXEC     A1
 MECAFF   SYNONYM  A1
 MECAFF-S SYNONYM  A1
END-OF-FILE OR END-OF-TAPE
```

## 4.2 Logical layering of the CMS side sources

The sources making up the CMS side of MECAFF can be seen as logically organized in the following (not necessarily non-overlapping) layers, starting from the bottom:

- MECAFF-API and -tools layer
  Interface:
  ```
  BOOL.H
  FSIO.H
  ```
  Implementation:
  ```
  WR3270.ASSEMBLE
  FSIO.C
  FS-QRY.C
  FS-CTL.C
  FSRDTEST.C
  ```
- 3270 stream encoding/decoding layer
  Interface:
  ```
  FS3270.H
  AID3270.H
  ```
  Implementation:
  ```
  FS3270.C
  ```
- Support layer
  Interface:
  ```
  EEUTIL.H
  ```
  Implementation:
  ```
  EEUTL1.C
  EEUTL2.C
  ```
- EE  core layer
  Interface:
  ```
  EECORE.H
  EESCRN.H
  ```
  Implementation:
  ```
  EECORE.C
  EESCRN.C
  ```
- EE tools layer
  Interface:
  ```
  EEMAIN.H
  IND$FILE.H
  ```
  Implementation:
  ```
  EEMAIN.C
  EECMDS.C
  EEPREFIX.C
  EELIST.C
  EEHELP.C
  FSHELP C
  IND$DENC.C
  IND$SCRN.C
  IND$FILE.C
  ```

## 4.3 Building the MECAFF modules

The MECAFF tools are built using the following 3 EXECs:

- `CMSCOMP EXEC`
  this EXEC creates the TEXT files for a minimal GCCLIB subset required in combination with the PDPCLIB to link the static MODULEs.
  It copies the required files from the GCCLIB source disk (user GCCCMS on VM/370 SixPack 1.2) to minidisk A, assembles resp. compiles them and erases the source files from GCCLIB.
  This EXEC needs only to be executed initially or when the TEXT files were deleted.
- `FSCOMP EXEC`
  this EXEC conditionally assembles and compiles the MECAFF sources if the TEXT files are missing or older as the corresponding source file.
  (assembling is done with ASM EXEC, compiling with CC EXEC, both use the tool `IS-STALE` for the conditionally processing, which is compiled and linked if necessary)
- `FSLINK EXEC`
  this EXEC links all MECAFF programs: the dynamically linked programs have the filename given in the MECAFF-manual, the statically linked MODULEs have a filename ending with S.
  The following table gives the corresponding names of the generated static MODULEs:

| **dynamic module** | **static module** |
|---|---|
| `FS-QRY   MODULE` | `FS-QRY-S MODULE` |
| `FS_CTL   MODULE` | `FS_CTL-S MODULE` |
| `EE       MODULE` | `EE-S     MODULE` |
| `FSHELP   MODULE` | `FSHELP-S MODULE` |
| `FSRDTEST MODULE` | `FSRDTSTS MODULE` |
| `IND$FILE MODULE` | `IND$FILS MODULE` |

## 5 Java-Sources of the external MECAFF process

The external MECAFF process was developed with an Eclipse Java project using Eclipse Ganymede Version 3.4.1.

The Java-sources-ZIP file has the complete content of the project directory (except the `bin`-directory with the compilation results), including the generated Javadoc for the project.

To add the MECAFF Java project to an Eclipse workspace:

- unpack the Java-sources-ZIP file in a new empty directory, preferably in the workspace base directory
- in Eclipse, invoke the *Import…* function from the background context menu in the Package Explorer, then “Existing project into Workspace”, select the directory above and then *Finish*.
- Compile the sources with the menu *Project* >> *Clean…* and then selecting the project just imported.

The jar-file for MECAFF can be created by selecting the file `mecaff_jar.jardesc` in the Package Explorer tree and invoking *Create JAR* in the context menu of this file. The file `mecaff.jar` will be created in the project directory.

The Javadoc can be browsed by opening the `index.html` file in the `doc` subdirectory of the project. The ant-file `javadoc.xml` can be used to regenerate the Javadoc using the *External tools* menu-button (add an entry with *External Tools Configuration…*).