Emil

a message converter

Version 1.1

26 november 1993

Martin Wendel
UDAC

1. INTRODUCTION

Emil is a tool for converting between message formats used by MIME, Eudora, SUN mailtool, PC and Mac based mail clients, etc. The purpose of Emil is to facilitate a smooth transition to MIME for Internet Mail on a national basis. It is to be regarded as an intermediate tool kit mainly to be used until the transition to MIME is complete.

Emil, correctly configured, will enable a site with a heterogeneous mail environment, consisting of various types of mail clients, to send and receive mail as if it were a homogenous mail environment; for instance sending only MIME messages to the outside world.

Emil is controlled by a few TCL scripts that handles headers and controls conversion by programming the command data structure used to perform the conversion. Emil can be compiled either to function on it's own as a standalone argument driven message conversion filter, or to be linked with IDA sendmail or Sendmail8.6.4 to form a powerful message conversion gateway. Either way, Emil will use the same TCL scripts to control conversion and header handling.

When installing Emil you'll get some basic scripts that will handle MIME, SUN mailtool, straight rfc822 with BinHex and straight rfc822 with uuencode. The enclosed scripts should not be viewed as a complete conversion environment, but as a crude set-up that can be used as a base for further development and fine tuning of the configuration.. I urge you to study them **closely** before conducting any changes to them or implementing new scripts.

The latest release of Emil is available by anonymous ftp at **ftp.uu.se** (under `/pub/unix/networking/mail/emil`).

2. INSTALLATION

First of all, Emil uses TCL as a configuration language. Get TCL from your local FTP site and install it (it shouldn't be to hard to make it work), use version 7.0 or later. From now on the directory which holds libtcl.a will be called TCLLIB and the directory which hold tcl.h will be called TCLINCLUDE. The directory containing the Emil source will be called EMILDIR.

Emil has been successfully installed on the following systems: AIX3.1, HPUX9, SunOS4, SunOS5, Ultrix4.3. If you're able to install IDA sendmail or Sendmail8.6.4 you shouldn't have much problems with Emil. I've managed to keep it quite simple and not used to much system dependant code. It's primarily just a lot of string manipulation.

2.1. Options

First check the paths and flags in EMILDIR/Makefile. These are set to reasonable defaults and need not be changed unless you've installed TCL in a non default location or want Emil installed in i non default location.

`DODEBUG` - set to `-DEMILDEBUG` if you want debugging (recommended).

`LOGLEVEL` - set `LOGLEVEL` to `LOG_DEBUG` if you want copious logging, otherwise use `LOG_ERR`. A good choice is to start out with `LOG_DEBUG` and then change to `LOG_ERR` when everything seems stable.

`LOGFILE` - set to the path of the logfile to use.

`SPOOLDIR` - set to where standalone Emil will spool it's working files.

`CFLAGS` - set include path to where tcl.h is located.

`LDFLAGS` - set search path to where libtcl.a is located.

`prefix` - relative to this everything will be installed.

2.2. Emil Linked into Sendmail or Emil Standalone

If using Emil linked with sendmail, sendmail will be able to act as a MIME gateway, converting messages to and from the MIME specification. If using Emil without sendmail, it'll work as a filter, converting messages as called. Beware that there is no way to use Emil as a mailer in the sense sendmail uses mailers.

2.2.1. If Compiling without Sendmail (Standalone)

- Edit EMILDIR/Makefile according to the above described options.
- Type `./configure.` This runs a configuration script created by GNU autoconf, which configures Emil for your system and creates `emakefile` and `config.h`.
- Type `make emil`. This will generate the Emil standalone filter.
- Type `make install` to install. The TCL routines used by Emil will be installed in `libdir/emil`, the charset file will be installed in `libdir`, the program `emil` will be installed in `bindir`. All of these are relative to `prefix`.

2.2.2. If Compiling with IDA Sendmail

- Unpack sendmail-5.67a+IDA-1.5. Put Emil-1.1.tar in SENDMAILSRC/ida and type `tar xf Emil-1.1.tar` to unpack it.
- Apply the included patchfile `Sendmail-IDA.patch` to the sendmail source. In SENDMAILSRC/src type `patch < Sendmail-IDA.patch`. If you don't have Larry Wall's patch program, get it!
- Edit EMILDIR/Makefile according to the above described options.
- Type `./configure.` This runs a configuration script created by GNU autoconf, which configures Emil for your system and creates `emakefile` and `config.h`.
- Type `make sendmailIDA`. This will generate the Emil library to be used with IDA sendmail.
- Type `make install` to install. The TCL routines used by Emil will be installed in `libdir/emil`, the charset file will be installed in `libdir`. All of these are relative to `prefix`.
- Edit SENDMAILSRC/src/conf.h according to sendmail's installation documentation and also make sure `EMIL` is defined.
- Edit SENDMAILSRC/src/Makefile according to sendmail's installation documentation.

  Append -I$TCLINCLUDE -I../ida/emil to CFLAGS

  Append -L$TCLLIB -L../ida/emil to LDFLAGS. If using SunOS5 also append -R/usr/lib:/usr/ucblib to LDFLAGS.

  Append -lEmil -tcl -lm to LIBS:

Follow the instructions with the sendmail distribution to compile and install IDA sendmail.

2.2.3. If Compiling with Sendmail8.6.4

- Unpack Sendmail8.6.4. Put Emil-1.1.tar in SENDMAILSRC and type `tar xf Emil-1.1.tar` to unpack it.

- Apply the included patchfile `Sendmail8.6.4.patch` to the `src` directory of sendmail. In SENDMAILSRC/src type `patch < Sendmail8.6.4.patch`. If you don.t have Larry Walls patch program, get it.

- Edit EMILDIR/Makefile according to the above described options.

- Type `./configure.` This runs a configuration script created by GNU autoconf, which configures Emil for your system and creates `emakefile` and `config.h`.

- Type `make sendmail8`. This will generate the Emil library to be used with IDA sendmail.

- Type `make install` to install. The TCL routines used by Emil will be installed in `libdir/emil`, the charset file will be installed in `libdir`. All of these are relative to `prefix`.

- Go to SENDMAILSRC and edit the Makefile.

  Add -I/TCLINCLUDE -I../emil to INCDIRS.

  Add -L/TCLLIB -L../emil to LIBDIRS.

  Add -lEmil -ltcl -lm to LIBS.

- Make sure `EMIL` is defined in `conf.h`.

Follow the instructions with the sendmail distribution to compile and install Sendmail8.6.4.

3. CONFIGURING AND USING EMIL

The distribution is provided a few TCL scripts that can be used as is, or be used as a base for further configuration. Enclosed with the scripts is also a readme file describing the function and conversions they apply.

3.1. Configuring and Using Emil Standalone

Little configuration is needed when using Emil in standalone mode. Emil functions as an argument driven filter reading input either from a file or stdin, copying input to a temporary file and writing output to a file or stdout. Usage:

> **emil** [**-i** *Infile*] [**-o** *Outfile*] [**-c** *Input_Charset*] *Conversion_Script*

If *Infile* is not specified emil will use stdin. If *Outfile* is not specified Emil will use stdout. If *Input_Charset* is not specified, be sure that charset is somehow specified in the incoming message in a way recognised by the TCL scripts in EMILLIB/headers or no charset conversion will be possible. The *Input_Charset* is used as default for incoming messages, if not specified in the message itself.

Conversion to the target format is specified in the TCL script EMILLIB/convert/*Conversion_Script*.tcl.

For instance, converting the file `input_message`, which contains an uuencoded file and some text written in ISO-8859-1, to a MIME message, type:

```
emil -i input_message -c iso-8859-1 mime
```

The MIME message will be sent to standard output. To save it in a file named mime_message, type:

```
emil -i input_message -o mime_message -c iso-8859-1
mime
```

Remember, the charset specified with the c flag is the character set you suspect the original message is written in. What character set to be used in the output is specified in the TCL conversion script being used.

3.2. Configuring and Using Emil with Sendmail

When using Emil with sendmail, your sendmail.cf file needs to be configured for Emil to be usable. Like Emil standalone, a `convert_script` needs to be specified for each mailer using Emil. Additionally a default charset should be specified. The default charset is used when no charset is specified in the incoming message.

The `convert_script` is specified with the mailer field `T` in sendmail.cf. Default charset is specified with the field `C` along with an `X` field for the default char escape (see the charset documentation for a complete description of suitable values to the `C` and `X` fields).

For example, a TCP mailer in IDA sendmail using character set ISO-8859-1 and wanting MIME format messages, could be specified as:

```
MTCP,P=[IPC],F=CDFMXhnmu,E=\r\n,R=10/11,S=11,A=IPC,
C=ISO-8859-1,X=29,T=mime
```

By adding these `C`, `X` and `T` fields to any mailer, charset translation and Emil conversion is enabled. In Sendmail8.6.4 the corresponding mailer could be specified as:

```
Msmtp,    P=[IPC], F=mDFMuX, S=11/31, R=21, E=\r\n
          A=IPC $h, C=ISO-8859-1, X=29, T=mime
```

While charset specification with the `C` field is case insensitive, conversion script specification with the `T` field is not. What is specified as `conversion_script` will be mapped directly to EMILLIB/convert/`conversion_script.tcl`. Make sure that the file exists, otherwise conversion will fail.

The above two examples will both result in sendmail sending MIME based messages with SMTP by default. If you instead leave the default SMTP mailers as they come, and modify some new mailers, you can selectively perform message conversion on hosts specified in the mailertable, using these new mailers. The above two mailers could for instance be given the name MIME.

3.2.1. Adding Support for Local User Based Conversion

When all users on a host should get the same conversion, simply add an appropriate conversion script to the local mailer (similar to the example above for the TCP mailer).

When Emil should convert messages differently for different users on a single host, i.e. there is no single qualified conversion scheme for the local mailer, a different approach could be used:

1. Create the local mailers needed. A good practice is to choose mailer names beginning with the string "local" (for example: local-mime, local-mac etc.).

2. Add one pseudonym for the local host in sendmail.cf for each local mailer.

3. Specify in mailertable, local mailer/pseudonym pairs as appropriate.

4. Route the local users mail through one of the pseudonyms (this could be done in their .forward files as user@pseudonym).

At our site we use a slightly different approach. The method described above yields the local mailer as sender mailer when one of the users sends a message, thus there is no way to differentiate the character set used on a mailer basis. To be able to accomplish that, one needs to couple mailer/user directly, i.e. no routes. I've chosen to do this by adding another sendmail table that specifies mailer/user pairs in much the same way as mailertable. *In addition, the rules resolving local mailer in* sendmail.cf *needs to be modified to make this new local mailer table to be consulted. This is not a standard measure and I must discourage inexperienced administrators to follow this method. See Appendix B for a more detailed description of these matters.*

4. GENERAL OPERATION

Emil is a message converter, and as such it uses conversion routines and control routines. The conversion routines are written in C and are adjusted to work in line mode rather than character mode. This is intended to fit in with sendmail. The control routines are new TCL commands written in C and linked with the application. The scripts resides under LIBDIR/emil hereafter called EMILLIB.

Simply put, Emil operates in three passes:

1. The investigation pass - In which the message is read and the contents are investigated.

2. The process pass - In which the gathered data is processed and a scheme for conversion is created.

3. The conversion pass - In which the conversion scheme is applied and the message is sent.

4.1. The Investigation Pass

To be able to get information about the message to be converted Emil parses the message and while doing so constructs a linked list of `MessageStruct` objects (look in the file `mkspec.h` for the definition of `MessageStruct`). When integrated with sendmail, Emil will parse the message during sendmail's collect phase. When standalone, Emil will parse the message while dumping it to a temporary file.

When found, Emil will check encoded parts of the message by doing a test decode of them. This is necessary to be able to retrieve content length, file type and file name, as well as to be able to check validity of the encoding. The initiative to decode must, however, come from the TCL control routines. Thus, the test decode will only take place if instructed to do so. (Emil can and will check for the BinHex and uuencode preamble automatically only if instructed to do so, and if it is found will initiate the test decode of the encoding.)

The objects of type `MessageStruct` contains a description of each sequential homogenous body part of the message. The main header is described in the first `MessageStruct` and thereafter follows MessageStructs for each body part containing text, attachments etc. The `MessageStruct` contains information like `type`, `encoding`, `start_line` (line count of start), `end_line` (line count of end), `filename`, `charset` and so on (figure 1).

When integrated with sendmail, `start_line` begins at the body of the message. When standalone, `start_line` begins at the start of the main headers. Header lines are stored in a linked list of HeaderStructs that are attached to the MessageStruct and are not treated as being part of the body part, hence the value of end_line of the first MessageStruct set to -1 (figure 1).

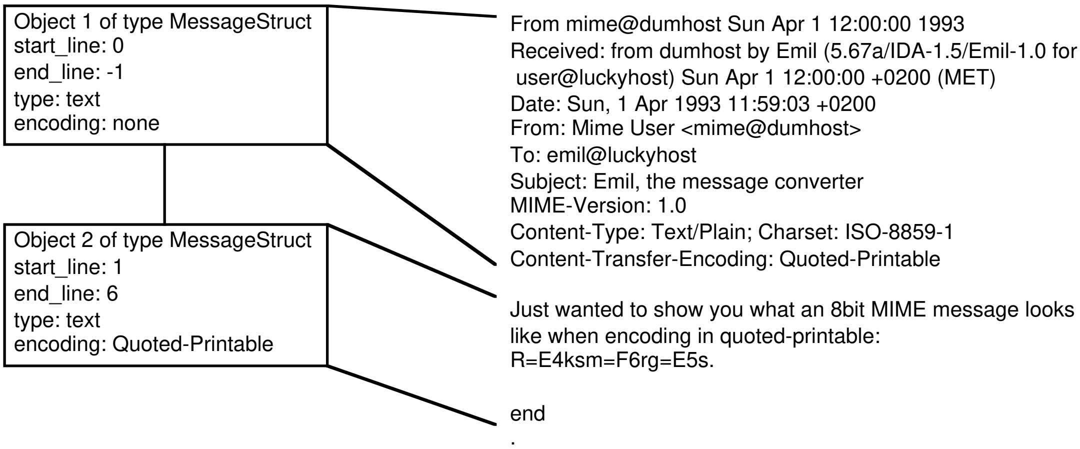

Figure 1. A simple message as described by the MessageStruct list.

Emil launches MKLIB/header.tcl for each body part being investigated, including the main message header part. It is launched after the headers (if any) of a body part has been read and loaded. The script contains a list of which investigation scripts under MKLIB/headers should be sourced. The investigation scripts mainly sets type and encoding of the body part by analysing the gathered header lines for the body part.

In the case of a MIME message, the investigation script MKLIB/headers/mime.tcl triggers a test decode from the information extracted from the headers. The script also sets boundaries in case of a Multipart message, character set in case of text, application type in case of an application etc.

Just after the end of a body part has been detected (for example the end of an attachment) the script MKLIB/check/settype.tcl is launched. This script will set type and subtype of BinHex and uuencode attachments.

4.2. The Process Pass

When the entire message is spooled and the investigation pass has gathered all relevant information, it is time to resolve the recipient's address and a relay host. This is done by sendmail as usual (if linked with sendmail).

For each recipient mailer, Emil allocates a fresh TCL stack, copies the main header structure (if more than one recipient; This is to ensure that changes to the headers applied by Emil will affect only the current recipient) and launches the script MKLIB/run.tcl This script loads a small set of help routines used later.

Now the specific conversion script configured with the mailer is launched. Available scripts can be found under MKLIB/convert. These scripts parses the MessageStruct list and configures appropriate encode/decode commands for each body part, as well as does the necessary changes to the header structures.

Once the process pass is done, the list of MessageStruct contains a complete set of instructions for the conversion of the message format.

4.3. The Conversion Pass

The conversion pass involves applying the conversion instructions set up in the process pass and then delivering the message.

5. ERROR HANDLING

Emil uses a very simple error handling scheme: If errors are encountered Emil will revert to transparent mode, i.e. sending the message (either part of it or the whole of it) unconverted. Recoverable errors are not corrected. However, Emil distinguishes between fatal and nonfatal errors. Detection of a nonfatal error in some part of a message simply results in conversion failure of that part. Emil will regard the specified type invalid and treat it as text. Detection of a fatal error will result in complete conversion failure and no conversion of that message will take part. It is important to understand the definition of fatal and nonfatal error in Emil.

Fatal errors include:

- Any syntactic error from TCL.
- Sending mail with an unconfigured mailer will be regarded as a fatal error.

Nonfatal errors include:

- An invalid encoding, i.e. checksum error, length error etc.
- An unknown character set.

Everything else unrecognised will be regarded as text and possibly a character set conversion will take place. For instance, because a BinHex file with a checksum error will be treated as text, it is important to be aware of what conversions can be applied to text by Emil.

6. EMIL SPECIFICS

6.1. Encodings

Emil will recognise encoded formats when triggered to do so by the TCL scripts. If not triggered Emil will regard the input as text. A violation of the encoding format will also make Emil regard the input as text. The currently recognised encodings are: BinHex, uuencode, Base64 and Quoted-Printable.

6.1.1. BinHex

Emil will recognise and check a BinHexed attachment whether containing an empty resource fork or not. However, Emil will only be able to expand the data fork, thus commanding Emil to HexBin an attachment containing a resource fork will result in the resource fork being junked. Emil will, though, specifically set type BINHEX on a BinHexed file with nonempty resource fork, whereas in the case of a data fork only file Emil will set the type from the type conversion tables in `MKLIB/check/settype.tcl.` Of course in both cases encoding will be set to BINHEX. I. e. a BinHexed attachment with non empty resource fork is recognised by having both encoding and type set to BINHEX. To make Emil look for BinHex in incoming messages the TCL command `setcheck "BinHex"` must be given once for every message.

When BinHexing a file Emil will stuff the file in the data fork and the resource fork will always be empty. If not specified, filename will be set to `noname`, type will be set to `TEXT` and creator will be set to `ttxt`. Although Emil will recognise, but not check, AppleSingle and AppleDouble while decoding uuencode and Base64 these will never be expanded but will always be regarded as an ordinary data fork file. It is therefore important to specifically not convert AppleSingle or AppleDouble files to BinHex. These formats should be checked for by the conversion TCL script.

6.1.2. Uuencode

Emil will recognise and check an uuencoded attachment. Emil will recognise if the file is an AppleSingle or AppleDouble file by checking the first four bytes for the AppleSingle or AppleDouble magic number. No further processing of expanding or checking the validity of the AppleSingle/AppleDouble file is conducted by Emil at the moment. To make Emil look for uuencode in incoming messages the TCL command `setcheck "uuencode"` must be given once for every message.

When uuencoding, filename will be set to `noname` if not specified while mode will always be set to 644. Routines for adding a corresponding extension to the filename are available.

6.1.3. Base64

Emil will recognise and check a base64 encoded attachment. Just as for uuencode Emil will check for an AppleSingle/AppleDouble magic number. On the whole Emil treats Base64 very much like uuencode when it comes to filenames, file types etc.

6.1.4. Quoted-Printable

Emil will recognise and check a text encoded in Quoted-Printable. When encoding Quoted-Printable line length will be kept close to 60 characters and every character less than `0x21` (!) or greater than `0x7F` (DEL) including `0x3D` (=) will be quoted.

6.2. Character Set Conversion

Emil uses a character set conversion routine `strncnv` developed by Keld Jørn Simonsen which is based on his work in RFC1345. The routine is very flexible and handles a vast number of character sets. Due to it's design there is a possibility that a character set conversion will produce some MNEMONIC characters when specific characters can not be matched in the target character set. Although possibly unwanted, these characters are unique and as such reversible (i.e. no information loss).

In addition to `strncnv`, Emil provides an 8 to 7 bit character conversion routine that maps 8 bit characters to 7 bit nonreversibly. This conversion is adapted to Scandinavian conditions and maps between ISO-8859-1 and ISO-646-SE.

6.3. File Names and File Types

6.3.1. File Names

Emil disallows the use of high 8 bit characters in file names. Characters accepted in file names are any character higher than SPACE, lower than '~' and excluding slash '/'. Characters not accepted will be set to underscore '_'.

6.3.2. File Types

File types are detected and set in three ways:

1. By file name extensions. Emil saves the file name extension and the file name separately. The file name extension can be retrieved from TCL, by the command `getextension`, and can be used to set file type accordingly to what's configured. When encoding, file name extension can be set by the command `setextension`.

2. By type and creator. When decoding a BinHexed file, Emil will save type and creator. These can be retrieved from TCL, by the commands `getbinhextype` and `getbinhexauth`, and can be used to set file type accordingly to what's configured. When encoding BinHex, type and creator can be set by the commands `setbinhextype` and `setbinhexauth`.

3. By header lines. When investigating a MIME message, Emil can retrieve the Content-Type headers and thus gain knowledge of file type. When converting to a MIME message, Content-Type header lines should be set according to file type.

6.3.3. File Names Revisited

During the process pass, Emil sets file name extension, BinHex type and creator according to file type and target encoding. If no file name is available, Emil sets `noname` as default file name. The three file encodings are treated differently:

BinHex - If type and creator is set, Emil uses them and sets file name without extension. If type and creator is not set, Emil adds the extension to file name and sets as default type to `TEXT` and creator to `ttxt`.

Uuencode - If extension is set it will be appended to file name.

Base64 - If extension is set it will be appended to file name.

6.4. Multipart Messages

When investigating a Multipart message, start of a new part can be triggered either by a boundary or by a preamble of a BinHexed or uuencoded attachment. If a boundary is found, Emil will trigger a new MessageStruct.

When processing a Multipart message, the command `dosubheaders` will enable output of boundaries and subheaders in front of each body part as specified.

6.5. Queue Handling with Sendmail

When compiled with sendmail, Emil will queue the MessageStruct list just after the investigation pass is completed. Emil's queue files are named `mf[Message ID]`. When sendmail enters deliver mode, Emil will read the queue file and rebuild the MessageStruct list. This ensures full operability even when delivery fails.

7. EMIL'S CONTROL SCRIPTS

As mentioned, Emil uses TCL scripts to control conversion and header handling. It is fairly easy to alter/extend functionality by modifying these scripts or by adding new ones. The new TCL commands added by Emil are ones that will change MessageStruct variables and add pointers to functions that will perform conversion. Important variables are: type, encoding, subtype, fcharset and tcharset.

type - is one of the predefined major types of MIME and a few more: AppleDouble, AppleSingle, Application, Audio, BinHex, External, Image, Message, Multipart, Text and Video.

encoding - is one of: 7bit, 8bit, Base64, BinHex, Binary, Quoted-Printable and uuencode. Binary is rarely used if not set in the input message.

subtype - is just a string. This string is fully exploited by the TCL scripts to save image type, application type and so on. Subtype is not to be mistaken for the MIME subtype, the implication of subtype in Emil is much more extensive than in MIME.

fcharset - is the source character set of an incoming text body part.

tcharset - is the target character set that outgoing text body parts should use.

7.1. The Investigation Pass Scripts

Starting out with an example; figure 2 shows the initial investigate script sourced by MKLIB/header.tcl. This script is run for each MessageStruct created during the investigation pass (each body part). In short it increments the variable PART_NUMBER each time run. If run for the first time it checks for the header line `Content-Conversion: prohibited` (as specified by RFC1344) and if found sets Emil in transparent mode by issuing the command `dotunnel`, else it activates Emil's check for a BinHex or uuencode preamble.

```
#
# header/init.tcl
#################
#
if {[is_set "PART_NUMBER"]} {
 set PART_NUMBER [expr $PART_NUMBER+1]
} \
else \
{
 set PART_NUMBER 0
 set CONTCONV [getheader "Content-Conversion"]
 if {[is_set CONTCONV]} {
  set CONTC [string toupper $CONTCONV]
  if {$CONTC == "PROHIBITED"} {
   dotunnel
   return
  }
 }
}
setcheck "uuencode"
setcheck "binhex"
```

Figure 2. The investigation pass script init.tcl.

The script mnemonic.tcl (figure 3) checks for the header lines X-Charset and X-Char-Esc as used by IDA sendmail compiled with BIT8 defined. If found and if message is not found to be a MIME message, fcharset is set to what's specified in the header.

The scripts mime.tcl and mailtool.tcl are much more extensive than these two and I urge you to study them closely to gain full understanding of what is set in the investigation pass.

```
#
# header/mnemonic.tcl
#####################
#

proc check_header_mnemonic {} {
 global MAINTYPE
 global SPECHEADER
 global MNEMESCAPE
 if {[is_set MAINTYPE]} {
  if {$MAINTYPE == "MIME"} {
   return 0
  }
 }
 if {[setfromcharset [getheader X-Charset]] != 0} {
  addspecheader X-Charset
  if {[set MNEMESCAPE [getheader X-Char-Esc]] != 0} {
   addspecheader X-Char-Esc
  }
 }
}
```

Figure 3. The investigation pass script mnemonic.tcl.

After the end of a body part has been detected, either by a match of boundary or by the end of an encoding, Emil will launch the script MKLIB/check/settype.tcl (the procedure `check_binhex` can be seen in figure 4). This script will set type and subtype of BinHexed or uuencoded attachments. These will be set according to author and creator for BinHex and according to the file name extension for uuencode. New subtypes are easily added if necessary.

```
proc check_binhex {} {
 if {[getencoding] != "BINHEX"} {
  return 0
 }
 if {[gettype] == "BINHEX"} {
  return 0
 }
 # Default
 set MAC_TYPE [getbinhextype]
 set MAC_AUTH [getbinhexauth]
 settype "APPLICATION"
 setsubtype $MAC_TYPE
 debug "Grabbing MAC TYPE $MAC_TYPE"
 debug "Grabbing MAC AUTH $MAC_AUTH"
 case $MAC_AUTH in {
 MSWD {
   settype "APPLICATION"
   setsubtype "MS-WORD"
  }
 MACA {
   settype "APPLICATION"
   setsubtype "MAC-WRITE"
  }
 XCEL {
   settype "APPLICATION"
   setsubtype "EXCEL"
  }
 }
# Generic formats
 case $MAC_TYPE in {
 GIFf {
   settype "IMAGE"
   setsubtype "GIF"
  }
 JFIF {
   settype "IMAGE"
   setsubtype "JPEG"
  }
 JPEG {
   settype "IMAGE"
   setsubtype "JPEG"
  }
 }
}
```

Figure 4. `check_binhex` from the investigation pass script settype.tcl.

7.2. The Process Pass Scripts

The first thing done during the process pass is to launch MKLIB/run.tcl. This script will source a help script, check.tcl, with general routines. If new scripts with general routines are added these should be placed under MKLIB/check and sourced by run.tcl.

The mailer that will actually deliver the message is defined with an Emil conversion script. The script is run immediately before delivery of the message and will parse the MessageStruct list and set appropriate conversion by issuing selected *decode* and *encode* commands. It will also add or delete necessary header lines. It is important when writing conversion scripts to be careful with what is converted and what is not. For example converting a BinHexed attachment with nonzero size resource fork, will definitely junk the resource fork. Therefore it is important to check for this kind of BinHex attachment by checking the type which is BINHEX when containing a resource fork.

The simplest conversion script is the transparent tunnel.tcl script (figure 5). The command given will set Emil in transparent mode and will unable all conversion whatsoever.

```
#
# convert/tunnel.tcl
####################
dotunnel
```

Figure 5. The conversion pass script tunnel.tcl.

8. EXTENDING EMIL

Emil uses TCL scripts to control conversion and is therefore easily extendible. When modifying or adding TCL scripts to Emil, it is important to be sure not to include any errors. A fatal error during the investigation pass will disable all conversion for all messages passing through the gateway. A fatal error during the process pass will disable conversion for all message passing through the specific mailer. It could be a good idea to check the validity of your changes with Emil standalone version. The scripts included in the distribution could easily be used as models for your changes.

APPENDIX A, TCL COMMANDS

A.1. Debugging

A.1.1. debug

Syntax: `debug` *`String`*

Returns: nothing.

Scope: TCL.

Evaluates *`String`* and writes it to the MkLogFile. *`String`* is prefixed with the message `TCL-DEBUG` to mark that the debug message originates from the TCL routines.

Note: This command writes the debug message with log level LOG_DEBUG. It is therefore essential that Emil is compiled with MKLOGLEVEL defined to LOG_DEBUG.

A.2. Tests

A.2.1. hasbody

Syntax: `hasbody`

Returns: `0` or `1`

Scope: The current MessageStruct.

Returns `1` (TRUE) if `start_line` of the current MessageStruct is less than or equal to `end_line`, i.e. the body part is not empty, otherwise `hasbody` returns `0` (FALSE). Beware that the smallest nonempty body part consists of a single linefeed.

A.2.2. is_multipart

Syntax: `is_multipart`

Returns: `0` or `1`

Scope: The current message.

Returns `1` (TRUE) if the current message is marked as a Multipart message, i.e. the length of the MessageStruct list is more than two, otherwise `is_multipart` returns `0` (FALSE).

Note: Emil normally detects this when allocating the third MessageStruct of any single message, thus Multipart need not be specified explicitly during the investigation pass.

A.2.3. is_root

Syntax: `is_root`

Returns: `0` or `1`

Scope: The current MessageStruct

Returns `1` (TRUE) if the current MessageStruct is the first in the list of MessageStructs, otherwise `is_root` returns `0` (FALSE).

A.2.4. is_set

Syntax: `is_set` *`Varname`*

Returns: `0` or `1`

Scope: TCL

Returns `1` (TRUE) if *`Varname`* is a defined variable, otherwise `is_set` returns `0` (FALSE). This test can also be done by catching the error using the TCL command:

```
[[catch {set Varname}] == 1]
```

However I prefer to use a generic command for variable checking instead.

A.2.5. knowfromcharset

Syntax: `knowfromcharset`

Returns: `0` or `1`

Scope: The current MessageStruct

Returns `1` (TRUE)if `fcharset` is set, either locally in the current MessageStruct, or globally in the root MessageStruct, otherwise knowfromcharset returns `0` (FALSE). Used to check if `fcharset` is not set and should be set to some reasonable default character set. It could be important to check if type is text and if any high characters has been detected (see `gethighchars`).

A.3. Text Related Commands

A.3.1. gethighchars

Syntax: `gethighchars`

Returns: Integer

Scope: The current MessageStruct

Returns the number of high characters (> 127) detected. When not decoding, Emil counts characters with the 8:th bit set. The integer returned from `gethighchars` applies to the current MessageStruct only. Can be used to test if a text body part has 8 bits characters when setting a default `fcharset`.

A.3.2. setfromcharset

Syntax: `setfromcharset` *`CharsetName`*

Returns: Nothing

Scope: The current MessageStruct

Sets `fcharset` to *`CharsetName`*. Used when issuing the `tocharset` command for character set conversion. No check is done concerning the validity of *`CharsetName`*. If *`CharsetName`* isn't a valid character set name, conversion will fail and Emil will send the body part through unconverted.

A.3.3. setfromesc

Syntax: `setfromesc` *`EscapeNumber`*

Returns: Nothing

Scope: The current MessageStruct

Sets character set conversion escape to *`EscapeNumber`*. Used by Keld Jørn Simonsens strncnv routine. If not set Emil will use a cleverly chosen default value.

A.3.4. to7bit

Syntax: `to7bit`

Returns: Nothing

Scope: The current MessageStruct

Adds 8 to 7 bit nonreversible character conversion to the conversion function list. Mapping of 8 bit characters to 7 bit characters are mainly used in the Scandinavian countries. `to7bit` does a translation from ISO-8859-1 to ISO-646-SE. The mapping table is static, Emil needs to be recompiled for changes on the mapping table to take effect.

A.3.5. tocharset

Syntax: `tocharset` *`CharsetName`*

Returns: `0` or `1`

Scope: The current MessageStruct

Sets `tcharset` to *`CharsetName`* and adds character conversion from `fcharset` to `tcharset` to the conversion function list. If either fcharset or tcharset are invalid no conversion function will be added and the text will be sent unconverted with an unknown character set. Returns `1` (TRUE) on success, otherwise `0` (FALSE).

A.4. Information Related Commands

A.4.1. getbinhexauth

Syntax: `getbinhexauth`

Returns: String

Scope: The current MessageStruct

Returns the author string of a BinHexed attachment as stored in the current MessageStruct. On failure `getbinhexauth` returns `0`.

A.4.2. getbinhextype

Syntax: `getbinhextype`

Returns: String

Scope: The current MessageStruct

Retrieves the type string of a BinHexed attachment as stored in the current MessageStruct. On failure `getbinhextype` returns `0`.

A.4.3. getextension

Syntax: `getextension`

Returns: String.

Scope: The current MessageStruct

Returns the extension of filename, as stored in the current MessageStruct, i.e. the characters after the last dot in a filename. On failure `getextension` returns `0`.

A.4.4. getfilename

Syntax: `getfilename`

Returns: String

Scope: The current MessageStruct

Returns filename as stored in the current MessageStruct. On failure `getfilename` returns `0`.

A.4.5. joinextension

Syntax: `joinextension`

Returns: Nothing

Scope: The Current MessageStruct

If `extension` and `filename` both are represented in the current MessageStruct, `filename` will be appended the contents of `extension` and `extension` will be nullified. Otherwise no action will be taken.

A.4.6. setbinhexauth

Syntax: `setbinhexauth` *`Auth`*

Returns: Nothing

Scope: The current MessageStruct

Sets the Auth bytes of a BinHexed attachment to *`Auth`*. If *`Auth`* is not exactly four bytes it is regarded a fatal error and the message will be sent unconverted.

A.4.7. setbinhextype

Syntax: `setbinhextype` *`Type`*

Returns: Nothing

Scope: The current MessageStruct

Sets the type bytes of a BinHexed attachment to *Type. If* *`Type`* is not exactly four bytes it is regarded a fatal error and the message will be sent unconverted.

A.4.8. setextension

Syntax: `setextension` *`Extension`*

Returns: Nothing

Scope: The current MessageStruct

Sets extension of the current MessageStruct to *`Extension`*.

A.4.9. setfilename

Syntax: `setfilename` *`Name`*

Returns: Nothing

Scope: The current MessageStruct

Sets filename of the current MessageStruct to *`Name`*.

A.5. Headers and Boundaries

A.5.1. addboundary

Syntax: `addboundary` *`Boundary`*

Returns: Nothing

Scope: The current message

Sets `boundary` of the current message to *`Boundary`*.

A.5.2. addendboundary

Syntax: `addendboundary` *`Boundary`*

Returns: Nothing

Scope: The current message

Sets `endboundary` of the current message to *`Boundary`*.

A.5.3. addheader

Syntax: `addheader` *`HField HValue`*

Returns: Nothing

Scope: The current MessageStruct

Adds the header *`HField`* with value *`HValue`* to the header list of the current MessageStruct.

A.5.4. addspecheader

Syntax: `addspecheader` *`HField`*

Returns: Nothing

Scope: The current MessageStruct

Adds the header *`HField`* to the list of special headers which might be removed by the command `rmspecheader`.

A.5.5. makeboundary

Syntax: `makeboundary`

Returns: String

Scope: The current message

Returns a boundary string that could be regarded unique.

A.5.6. putendboundary

Syntax: `putendboundary`

Returns: nothing

Scope: The current MessageStruct

Marks that the end boundary of the current boundary should be printed at the end of this body part.

A.5.7. getheader

Syntax: `getheader` *`HField`*

Returns: String

Scope: The current MessageStruct

Returns the value of header *`HField`* of the current MessageStruct. On failure `getheader` returns `0`.

A.5.8. listheaders

Syntax: `listheaders`

Returns: List

Scope: The current MessageStruct

Returns the header fields of the current MessageStruct as a TCL list. If no headers are available `listheaders` returns `0`.

A.5.9. rmheader

Syntax: `rmheader` *`HField`*

Returns: Nothing

Scope: The current MessageStruct

Removes the header *`HField`* from the header list of the current MessageStruct.

A.5.10. rmspecheader

Syntax: `rmspecheader`

Returns: Nothing

Scope: The current MessageStruct

Removes all the *`Hfields`* in the list of special headers from the header list of the current MessageStruct.

A.6. Types and Encodings

A.6.1. getencoding

Syntax: `getencoding`

Returns: String

Scope: The current MessageStruct

Returns the encoding of the current MessageStruct. Encoding is one of 7BIT, 8BIT, BINARY, BASE64, QUOTED-PRINTABLE, UUENCODE or BINHEX.

A.6.2. getmaintype

Syntax: `getmaintype`

Returns: String

Scope: The current message

Returns the main type of the current message as set by `setmaintype`. Main type is a string and can be set to anything. Make sure to be consistent with case when using this variable.

A.6.3. getsubtype

Syntax: `getsubtype`

Returns: A string

Scope: The current MessageStruct

Returns the subtype of the current MessageStruct as set by setsubtype. Sub type is a string and can be set to anything. Make sure to be consistent with case when using this variable.

A.6.4. gettype

Syntax: `gettype`

Returns: A string

Scope: The current MessageStruct

Returns the type of the current MessageStruct. Type is one of: TEXT, APPLICATION, IMAGE, AUDIO, VIDEO, MULTIPART, MESSAGE; EXTERNAL, BINHEX, APPLESINGLE or APPLEDOUBLE.

Type BINHEX indicates the current MessageStruct represents a BinHexed attachment with a nonempty resource fork. Don't decode this attachment or you'll end up junking the resource fork.

Type APPLESINGLE indicates the current MessageStruct represents an AppleSingle attachment. The only reasonable encodings are Base64 and uuencode.

Type APPLEDOUBLE indicates the current MessageStruct represents the header/resource fork part of an AppleDouble file. Beware that the data fork will not be of type APPLEDOUBLE.

A.6.5. setencoding

Syntax: `setencoding` *`String`*

Returns: Nothing

Scope: The current MessageStruct

Sets the encoding of the current MessageStruct to *`String`*. *`String`* must be one of: 7BIT, 8BIT, BINARY, BASE64, QUOTED-PRINTABLE, UUENCODE or BINHEX, otherwise encoding will be set to UNKNOWN (NULL). The only thing modified by `setencoding` is the value of encoding. To actually encode or decode use the commands `encode` or `decode`, respectively.

A.6.6. setmaintype

Syntax: `setmaintype` *`String`*

Returns: Nothing

Scope: The current message

Sets main type of the current message to *`String`*.

A.6.7. setsubtype

Syntax: `setsubtype` *`String`*

Returns: Nothing

Scope: The current MessageStruct

Sets subtype of the current MessageStruct to *`String`*. Subtype is used by `getsubtype` and can be exploited to resolve file name extension or BinHex type etc.

A.6.8. settype

Syntax: `settype` *`String`*

Returns: Nothing

Scope: The current MessageStruct

Sets type of the current MessageStruct to *`String`*. *`String`* must be on of: TEXT, APPLICATION, IMAGE, AUDIO, VIDEO, MULTIPART, MESSAGE; EXTERNAL, BINHEX, APPLESINGLE or APPLEDOUBLE, otherwise type will be set to UNKNOWN (NULL).

A.7. Control

A.7.1. encode

Syntax: `encode` *`Encoding`*

Returns: Nothing

Scope: The current MessageStruct

Adds to the functions list the function to encode the contents of the current MessageStruct with *`Encoding`*. *`Encoding`* must be one of: BASE64, BINHEX, QUOTED-PRINTABLE or UUENCODE, otherwise a fatal error will occur.

A.7.2. decode

Syntax: `decode`

Returns: Nothing

Scope: The current MessageStruct

Adds to the functions list the function to decode the contents of the current MessageStruct. Emil will determine what decoding function needs to be applied by looking at the value of `encoding` of the current MessageStruct.

A.7.3. dosubheaders

Syntax: `dosubheaders`

Returns: Nothing

Scope: The current message

Commands Emil to output the current message with boundaries and subheaders. If not issued only the headers in the first MessageStruct will be sent and no boundary will separate the body parts.

Note: This command must be issued when sending a Multipart message of MIME type or SUN Mailtool type.

A.7.4. dotunnel

Syntax: `dotunnel`

Returns: Nothing

Scope: The current message

Commands Emil to disable all conversion of the message. If issued during the investigation pass, every copy of the message will be sent unconverted. If issued during the process pass, the copy processed will be sent unconverted. Beware that there's no return from `dotunnel,` it cannot be nullified.

A.7.5. getpartlen

Syntax: `getpartlen`

Returns: The number of lines in the current MessageStruct.

Scope: The current MessageStruct

Returns the number of lines of the content of the current body part. The header lines are not counted.

A.7.6. insertmessnode

Syntax: `insertmessnode`

Returns: Nothing

Scope: The current message.

Inserts a new MessageStruct in the MessageStruct list just before the current MessageStruct. The new MessageStruct is made active as the current MessageStruct. Used mainly to insert a Multipart header inside the body of a MIME message.

A.7.7. killbody

Syntax: `killbody`

Returns: Nothing

Scope: The current MessageStruct

Removes the current MessageStruct from the list and makes the next MessageStruct active. `killbody` is only allowed if the test `hasbody` returns FALSE (0) and the current MessageStruct is **not** the root MessageStruct. Used mainly to remove empty text bodies between the two body parts of an uuencoded AppleDouble file.

A.7.8. nextmessnode

Syntax: nextmessnode

Returns: An integer

Scope: The current message

Sets the next item in the list of MessageStructs the current MessageStruct. Used during the process pass to parse the MessageStruct list. If issued during the investigation pass a fatal error will occur (this is a safety measurement to avoid confusing Emil as to what MessageStruct is processed).

A.7.9. setcheck

Syntax: `setcheck` *`Encoding`*

Returns: Nothing

Scope: The current message

Enables check for *`Encoding`* in message. Valid values of Encoding are: *`uuencode`* and *`BinHex`*.

A.7.10. setpartlen

Syntax: `setpartlen` *`Size`*

Returns: True on success, false on failure.

Scope: The current MessageStruct

Sets the number of lines of the content of the current body part, not counting the header lines, to *`Size`*. This is a way to force Emil not to split up the current body part even if it contains suspected boundaries etc.

A.7.11. topmessnode

Syntax: `topmessnode`

Returns: Nothing

Scope: The current message

Sets the current MessageStruct the first object in the list of MessageStructs.

APPENDIX B, TIDBITS

B.1. Information retrieval

To be able to perform any selective conversion on a message, Emil needs information describing the format of the message. This information can be such as character set, encoding, filename etc., without which there is no way for Emil to determine what conversion algorithm to apply. There are three ways for Emil to obtain this information, namely by extracting:

- input mailer information
- message header information
- message body information

B.1.1. Input Mailer Information

As of now only the variable `fcharset` is provided from the input mailer for use by Emil. This is done by specifying the C field of the mailer in sendmail.cf[1] just like when using the BIT8 with IDA sendmail. The normal value of the C field would be the name of a character set[2]. When using Emil it will be copied to the variable `fcharset` used by the TCL scripts. If the value of `fcharset` in the root MessageStruct is handled properly by the TCL scripts, this, however, can be exploited quite heavily for other purposes.

The retrieval of the `fcharset` string by sendmail is not a target for further processing until the TCL command `tocharset` is employed. That is, `fcharset`, and thus the C field of the input mailer, can be safely used for any purpose until the command `tocharset` is issued. Knowing this, the initial value of `fcharset` can be set to some string that is useful while processing the MessageStructs in the TCL scripts.

If your mailertable is set up to couple a mailer with a specific network, the value of the C field could be set to discern the originating network. In conjunction with the target network (as specified by the target conversion script) some conclusions could be drawn (you know the originator and you know the recipient), for instance a message sent within the country could imply some country specific character set.

If `fcharset` is used for something else than to explicitly specify the originating character set, you must define `fcharset` to the character set chosen to be used as from character set before using the character conversion routines.

B.1.2. Message Header Information

Descriptive headers is important when trying to get information about the incoming message. The best example of this is the MIME format. Emil lets you retrieve the information in the headers of the input message by using the TCL scripts.

[1]See chapter 3
[2]As defined in RFC1345

B.1.3. Message Body Information

The last way of obtaining information about the incoming message is to check the body for encoding information or type information. Emil uses some TCL functions that will let you acquire this information. The TCL function `getencoding` yields the encoding of the body part, while the TCL function `gettype` yields the type of the body part. If type is equal to `text` you can use the TCL function `gethighchars` to produce the number of 8bit characters encountered in the body part. If the result is 0, the text is written with 7bit characters, else it is written with some 8bit characters.

B.1.4. Collaboration of Methods

When using all of these three methods of information retrieval you can get virtually total control of the message conversions. For instance, if you can determine, by use of the mailer information, that a message is being sent within the country, you can by using body information draw the conclusion that if it contains text with 8bit characters it is a ISO-8859-1 text, while if it contains text with no 8bit characters it is a national variant of ISO-646 text. These methods are free for you to use when configuring your site.

B.2. Local User Based Conversion

Local user based conversion is a method that requires that the local users need different mailers for final delivery of the message. Much in the same way as that of using mailertable to select mailer for hosts, you can use a localtable to select mailer for local users. This is essential when you run a mail server with users using different mail clients.

B.2.1. Adding more than one local mailer

First of all you need to add and customise some local mailers to perform conversions you want. This is done in sendmail.cf. The safest method is to copy the current local mailer and modify the copies. In IDA sendmail they could look like:

```
Mlocal, P=/bin/mail,F=DFMlrmns,R=25/10,S=10,A=mail -d $u,C=ISO-8859-1,X=29,T=tunnel
Mlocal-mime,P=/bin/mail,F=DFMlrmns,R=25/10,S=10,A=mail -d $u,C=ISO-8859-1,X=29,T=mime
Mlocal-sun,P=/bin/mail,F=DFMlrmns,R=25/10,S=10,A=mail -d $u,C=ISO-8859-1,X=29,T=mailtool
Mlocal-mac,P=/bin/mail,F=DFMlrmns,R=25/10,S=10,A=mail -d $u,C=MACINTOSH,X=29,T=eudora
Mlocal-pc,P=/bin/mail,F=DFMlrmns,R=25/10,S=10,A=mail -d $u,C=CP437,X=29,T=tunnel
```

The tunnel-script specified in Mlocal is a script that will perform no conversion whatsoever. In sendmail8.6.4 the same mailers could look like:

```
Mlocal,           P=/bin/mail, F=lsDFMrmn, S=10, R=20/40,
                  A=mail -d $u, C=ISO-8859-1,X=29,T=tunnel
Mlocal-mime,      P=/bin/mail, F=lsDFMrmn, S=10, R=20/40,
                  A=mail -d $u, C=ISO-8859-1, X=29, T=mime
Mlocal-sun,       P=/bin/mail, F=lsDFMrmn, S=10, R=20/40,
                  A=mail -d $u, C=ISO-8859-1, X=29, T=mailtool
Mlocal-mac,       P=/bin/mail, F=lsDFMrmn, S=10, R=20/40,
                  A=mail -d $u, C=MACINTOSH, X=29, T=eudora
Mlocal-pc,        P=/bin/mail, F=lsDFMrmn, S=10, R=20/40,
                  A=mail -d $u, C=CP437, X=29, T=ms-mail
```

Note that the mailers above are examples, the only thing portable about hem are the C, T and X fields. You should copy your own local mailer and add these fields accordingly.

When the local mailers are defined, the remaining trick is to make sendmail recognise them and use them for the local users.

B.2.2. Coupled Local User/Local Mailer Based Conversion

This method involves an additional sendmail table and some modifications of the rulesets in sendmail.cf that will change the behaviour of sendmail mailer resolution. *Don't do this unless you're familiar with* sendmail.cf *and know exactly what you are doing.*

Add a new table to sendmail.cf, for simplicity call it `localtable`.

- With IDA sendmail it could look like this:

```
OKT/usr/local/lib/mail/localtable
```

- In Sendmail8.6.4 it could look like:

```
Klocaltable dbm /usr/local/lib/mail/localtable
```

In IDA sendmail modify rulesets S0 and S26 to use `localtable` when resolving the local mailer.

- In ruleset S0 change the lines:

```
R?<$*>$+                    $#LOCAL$@$w$:$>25<$1>$2     and deliver.
R<@localhost>,$~A$*  $#LOCAL$@$w$:$1$2           Deliver forced local
```

  to look like this:

```
R?<$*>$+             $#$(T$2$:LOCAL$)$@$w$:$>25<$1>$2   and deliver.
R<@localhost>,$~A$*  $#$(T$1$:LOCAL$)$@$w$:$1$2  Deliver forced local
```

- In ruleset S26 change the line:

```
R<>,$+ $#LOCAL $@$w $:$1           local user
```

  to look like this:

```
R<>,$+ $#$(T$1$:LOCAL$)$@$w$:$1    local user
```

- Edit the new table `localtable` according to:

```
local-mime    user1
local-sun     user2
```

  and so on. Add making of `localtable` in your Makefile:

```
localtable.dir: localtable
       ./dbm -LN make localtable localtable
```

- Be sure to use the -r flag with xalparse to relativize to localhost. Remake `localtable`, refreeze your sendmail.fc and restart sendmail.

In Sendmail8.6.4 modify ruleset S0 to make sure the sendmail searches localtable when resolving the local mailer.

- Replace every occurrence of :

```
$#local $: $1
```

  with

```
$#$(localtable $1 $:local$) $: $1
```

- Replace every occurrence of:

```
$#local $: @ $1
```

  with

```
$#$(localtable $1 $:local$) $: @ $1
```

- Edit the new table localtable according to:

```
local-mime    user1
local-sun     user2
```

- Make the table by issuing the command:

```
makemap dbm /usr/local/lib/localtable < /usr/local/lib/localtable
```

- Make sure you refreeze the sendmail.fc and restart sendmail.