The UUDeview Decoding Library

Frank Pilhofer

January 3, 2026

Abstract

The UUDeview library is a highly portable set of functions that provide facilities for decoding *uuencoded*, *xxencoded*, *Base64* and *BinHex*-Encoded files as well as for encoding binary files into all of these representations except BinHex. This document describes how the features of encoding and decoding can be integrated into your own applications.

The information is intended for developers only, and is not required reading material for end users. It is assumed that the reader is familiar with the general issue of encoding and decoding and has some experience with the "C" programming language.

This document describes version 0.5, patchlevel 20 of the library.

1 Introduction

1.1 Background

The Internet provides us with a fast and reliable means of user-to-user message delivery, using private email or newsgroups. Both systems have originally been designed to transport plain-text messages. Over the years, some methods appeared allowing transport of arbitrary binary data by "encoding" the data into plain-text messages. But after these years, there are still certain problems handling the encoded data, and many recipients have difficulties decoding the messages back into their original form.

It should be the job of the mail delivery agent to handle sending and rend receiving binary data transparently. However, the support of most applications is limited, and several incompatibilities among different software exists.

There are three common formats for encoding binary data, called *uuencoding*, *Base64* and *BinHex*. Issues are further complicated by slight variations of the formats, the packaging, and some broken implementations.

Further problems arise with multi-part postings, where the encoding of a huge file has been split up into several individual messages to ensure proper transfer over gateways with limited message sizes. Very few software is able to properly sort and decode the parts. Even nowadays, many users are at a loss to decode these kinds of messages.

This is where the UUDeview Decoding Library steps in.

1.2 The Library

The UUDeview library makes an attempt at decoding nearly all kinds of encoded files. It is supposed to decode multi-part files as well as many files simultaneously. Part numbers are evaluated, thus making it possible to re-arrange parts that aren't in their correct order.

No assumptions are made on the format of the input file. Usually the input will be an email folder or newsgroup messages. If this is the case, the information found in header lines is evaluated; but plain encoded files with no surrounding information are also accepted. The input may also consist of concatenated parts and files.

Decoding files is done in two passes. During the first pass, all input files are scanned. Information is gathered about each chunk of encoded data. Besides the obvious data about type, position and size of the chunk, some environmental information from the envelope of a mail message is also gathered if available.

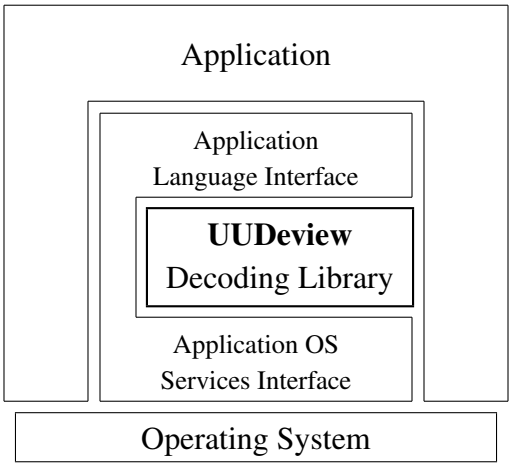


Figure 1: Integration of the Library

If the scanner finds a properly MIME-formatted message, a proper MIME parser steps into action. Because MIME messages include precise information about the message's contents, there is seldom doubt about its parts.

For other, non-MIME messages, the "Subject" header line is closely examined. Two informations are extracted: the part number (usually given in parentheses) and a unique identifier, which is used to group series of postings. If the subject is, for example, "uudeview.tgz (01/04)", the scanner concludes that this message is the first in a series of four, and the indicated filename is an ideal key to identify each of the four parts.

If the subject is incomplete (no part number) or missing, the scanner tries to make the best of the available information, but some of the advanced features won't work. For example, without any information about the part number, it must be assumed that the available parts are in correct order and can't be automatically rearranged.

All the information is gathered in a linked list. An application can then examine the nodes of the list and pick individual items for decoding. The decoding functions will then visit the parts of a file in correct order and extract the binary data.

Because of heavy testing of the routines against real-life data and many problem reports from users, the functions have become very robust, even against input files with few, missing or broken information.

Figure 1 displays how the library can be integrated into an application. The library does not assume any capabilities of the operating system or application language, and can thus be used in almost any environment. The few necessary interfaces must be provided by the application, which does usually know a great deal more about the target system.

The idea of the "language interface" is to allow integration of the library services into other programming languages; if the application is itself written in C, there's no need for a separate interface, of course. Such an interface currently exists for the Tcl scripting language; other examples might be Visual Basic, Perl or Delphi.

1.3 Terminology

These are some buzzwords that will be used in the following text.

- "Encoded data" is binary data encoded by one of the methods "uuencoding", "xxencoding", "Base64" or "BinHex".
- "Message" refers to both complete email messages and Usenet news postings, including the complete headers. The format of a message is described in [RFC0822]. A "message body" is an email message or news posting without headers.

- A “mail folder” is a number of concatenated messages.
- “MIME” refers to the standards set in [RFC1521].
- A “multipart message” is an entity described by the MIME standard. It is a single message divided into one or more individual parts by a unique boundary.
- A “partial message” is also described by the MIME standard. It is a message with an associated identifier and a part number. Large messages can be split into multiple partial messages on the sender’s side. The recipient’s software groups the partial messages by their identifier and composes them back into the original large message.
- The term “partial message” only refers to *one part* of the large message. The original, partialized message is referred to as “multi-part message” (note the hyphen). To clarify, one part of a multi-part message is a partial message.

2 Compiling the Library

On Unix systems, configuration and compilation is trivial. The script `configure` automatically checks your system and configures the library appropriately. A subsequent “make” compiles the modules and builds the final library.

On other systems, you must manually create the configuration file and the Makefile. The configuration file `config.h` contains a set of preprocessor definitions and macros that describe the available features on your systems.

2.1 Creating `config.h` by hand

You can find all available definitions in `config.h.in`. This file undefines all possible definitions; you can create your own configuration file starting from `config.h.in` and editing the necessary differences.

Most definitions are either present or absent, only a few need to have a value. If not explicitly mentioned, you can activate a definition by changing the default `undef` into `define`. The following definitions are available:

2.1.1 System Specific

`SYSTEM_DOS` Define for compilation on a *DOS* system. Currently unused.

`SYSTEM_QUICKWIN` Define for compilation within a *QuickWin*[1] program. Currently unused.

`SYSTEM_WINDLL` Causes all modules to include `<windows.h>` before any other include file. Makes `uulib.c` export a `DllEntryPoint` function.

`SYSTEM_OS2` Causes all modules to include `<os2.h>` before any other include file.

2.1.2 Compiler Specific

`PROTOTYPES` Define if your compiler supports function prototypes.

`UUEXPORT` This can be a declaration to all functions exported from the decoding library. Frequently needed when compiling into a shared library.

`TOOLEXPORT` Similar to `TOOLEXPORT`, but for the helper functions from the replacement functions in `fptools.c`.

[1] The Microsoft compilers offer the *QuickWin* target to allow terminal-oriented programs to run in the Windows environment

2.1.3 Header Files

There are a number of options that define whether header files are available on your system. Don't worry if some of them are not. If a header file is present, define "HAVE_*name-of-header*": HAVE_ERRNO_H, HAVE_FCNTL_H, HAVE_IO_H, HAVE_MALLOC_H, HAVE_MEMORY_H, HAVE_UNISTD_H and HAVE_SYS_TIME_H (for <sys/time.h>). Some other include files are needed as well, but there are no macros for mandatory include files.

There's also a number of header-specific definitions that do not fit into the general present-or-not-present scheme.

STDC_HEADERS Define if your header files conform to *ANSI C*. This requires that stdarg.h is present, that stdlib.h is available, defining both malloc() and free(), and that string.h defines the memory functions family (memcpy() etc).

HAVE_STDARG_H Implicitly set by STDC_HEADERS. You only need to define this one if STDC_HEADERS is not defined but <stdarg.h> is available.

HAVE_VARARGS_H *varargs* can be used as an alternative to *stdarg*. Define if the above two values are undefined and <varargs.h> is available.

TIME_WITH_SYS_TIME Define if HAVE_SYS_TIME_H and if both <sys/time.h> and <time.h> can be included without conflicting definitions.

2.1.4 Functions

HAVE_STDIO Define if standard I/O (stdin, stdout and stderr) is available.

HAVE_GETTIMEOFDAY Define if your system provides the gettimeofday() system call, which is needed to provide microsecond resolution to the busy callback. If this function is not available, time() is used.

2.1.5 Replacement Functions

The tools library fptools defines many functions that aren't standard on all systems. Most of them do not differ in behavior from their originals, but might be slightly slower. But since they are usually only needed in non-speed-critical sections, the replacements are used throughout the library. For a full listing of the available replacement functions, see section 11.

However, there are two functions, strerror and tempnam, that aren't fully implemented. The replacement strerror does not have a table of error messages and only produces the error number as string, and the "fake" tempnam does not necessarily use a proper temp directory.

Because some functionality is missing, the replacement functions should *only* be used if the original is not available.

strerror If your system does not provide a strerror function of its own, define to _FP_strerror. This causes the replacement function to be used throughout the library.

tempnam If your system does not provide a tempnam function of its own, define to _FP_tempnam. This causes the replacement function to be used throughout the library. Must not be defined if the function is in fact available.

2.2 Creating the Makefile by hand

The Makefile is automatically generated by the configuration script from the template in Makefile.in. This section explains how the template must be edited into a proper Makefile.

Just copy Makefile.in to Makefile and edit the place-holders for the following values.

CC Your system's "C" compiler.

CFLAGS The compilation flags to be passed to the compiler. This must include "-I." so that the include files from the local directory are found, and "-DHAVE_CONFIG_H" to declare that a configuration file is present.

RANLIB Set to "ranlib" if such a program is available on your system, or to ":" (colon) otherwise.

VERSION A string holding the release number of the library, currently "0.5"

PATCH A string holding the patchlevel, currently "20".

Some systems do not know Makefiles but offer the concept of a "project".[2] In this case, create a new project targeting a library and add all source codes to the project. Then, make sure that the include path includes the current directory. Add options to the compiler command so that the symbol "HAVE_CONFIG_H" gets defined. Additionally, the symbol "VERSION" must be defined as a string holding the release number, currently "0.5" and "PATCH" must be defined as a string holding the patch level, currently "20".

On 16-bit systems, the package should be compiled using the "Large" memory model, so that more than just 64k data space is available.

2.3 Compiling your Projects

Compiling the parts of your project that use the functions from the decoding library is pretty straightforward:

- All modules that call library functions must include the `<uudeview.h>` header file.
- Optionally, if you want to use the replacement functions to make your own application more portable, they may also include `<fptools.h>`.
- If your compiler understands about function prototypes, define the symbol `PROTOTYPES`. This causes the library functions to be declared with a full parameter list.
- Modify the include file search path so that the compiler finds the include files (usually with the "-I" option).
- Link with the `libuu.a` library, usually using the "-luu" option.
- Make sure the library is found (usually with the "-L" option).

3 Callback Functions

3.1 Intro

At some points, the decoding library offers to call your custom procedures to do jobs you want to take care of yourself. Some examples are the "Message Callback" to print a message or the "Busy Callback", which is frequently called during lengthy processing of data to indicate the progress. You can hook up your functions by calling some library function with a pointer to your function as a parameter.

In some cases, you will want that one of your functions receives certain data as a parameter. One reason to achieve this would be through global data; another possibility is provided through the passing of an opaque data pointer.

All callback functions are declared to take an additional parameter of type `void*`. When hooking up one of your callbacks, you can specify a value that will passed whenever your function is called. Since this pointer is never touched by the library, it can be any kind of data, usually some composed structure. Some application for the Message Callback might be a `FILE*` pointer to log the messages to.

For portability reasons, you should declare your callbacks with the first parameter actually being a `void*` pointer and only cast this pointer to its real type within the function body. This prevents compiler warnings about the callback setup.

[2] Actually, most project-oriented systems compile the project definitions into a Makefile for use by the back-ends.

3.2 Message Callback

For portability reasons, the library does not assume the availability of a terminal, so it does not initially know where to print messages to. The library generates some messages about its progress as well as more serious warnings and errors. An application should provide a message callback that displays them. The function might also choose to ignore informative messages and only display the fatal ones.

A Message Callback takes three parameters. The first one is the opaque data pointer of type `void*`. The second one is a text message of more or less arbitrary length without line breaks. The last parameter is an indicator of the seriousness of this message. A string representation of the warning level is also prefixed to the message.

UUMSG_MESSAGE This is just a plain informative message, nothing important. The application can choose to simply ignore the message. If a log file is available, it should be logged, but the message should never result in a modal dialogue.

UUMSG_NOTE "Note:" Still an informative message, meaning that the library made a decision on its own that might interest the user. One example for a note is that the setuid bit has been stripped from a file mode for security reasons. Notes are nothing serious and may still be ignored.

UUMSG_WARNING "Warning:" A warning indicates that a non-serious problem occurred which did not stop the library from proceeding with the current action. One example is a temporary file that could not be removed. Warnings should be displayed, but an application may decide to continue even without user intervention.

UUMSG_ERROR "ERROR:" A problem occurred that caused termination of the current request, for example if the library tried to access a non-existing file. After an error has occurred, the application should closely examine the resulting return code of the operation. Error messages are usually printed in modal dialogues; another option is to save the error message string somewhere and later print the error message after the application has examined the operation's return value.

UUMSG_FATAL "Fatal Error:" This would indicate that a serious problem has occurred that prevents the library from processing any more requests. Currently unused.

UUMSG_PANIC "Panic:" Such a message would indicate a panic condition, meaning the application should terminate without further clean-up handling. Unused so far.[3]

3.3 Busy Callback

Some library functions, like scanning of an input file or decoding an output file, can take quite some time. An application will usually want to inform the user of the progress. A custom "Busy Callback" can be provided to take care of this job. This function will then be called frequently while a large action is being executed within the library. It is not called when the application itself has control.

Apart from the usual opaque data pointer, the Busy Callback receives a structure of type `uuprogress` with the following members:

action What the library is currently doing. One of the following integer constants:

UUACT_IDLE The library is idle. This value shouldn't be seen in the Busy Callback, because the Busy Callback is never called in an idle state.

UUACT_SCANNING Scanning an input file.

UUACT_DECODING Decoding a file.

UUACT_COPYING Copying a file.

UUACT_ENCODING Encoding a file.

[3] It is not intended that this and the previous error levels will ever be used. Currently, there's no need to include handling for them.

curfile The name of the file we're working on. May include the full path. Guaranteed to be 256 characters or shorter.

partno When decoding a file, this is the current part number we're working on. May be zero.

numparts The maximum part number of this file. Guaranteed to be positive (non-zero).

percent The percentage of the current *part* already processed. The total percentage can be calculated as $(100 * partno - percent)/numparts$.

fsize The size of the current file. The percent information is only valid if this field is *positive*. Whenever the size of a file cannot be properly determined, this field is set to -1; in this case, the percent field may hold garbage.

In some cases, it is possible that the percent counter jumps backwards. This happens seldom enough not to worry about it, but the callback should take care not to crash in this case.[4]

The Busy Callback is declared to return an integer value. If a *non-zero* value is returned, the current operation from which the callback was called is canceled, which then aborts with a return value of `UURET_CANCEL` (see later).

3.4 File Callback

Input files are usually needed twice, first for scanning and then for decoding. If the input files are downloaded from a remote server, perhaps by *NNTP*, they would have to be stored on the local disk and await further handling. However, the user may choose not to decode some files after all.

If disk space is important, it is possible to install a "File Callback". When scanning a file, it is assigned an "Id". After scanning has completed, the application can delete the input file. If it should be required later on for decoding, the File Callback is called to map the Id back to a filename, possibly retrieving another copy and disposing of it afterwards.

The File Callback receives four parameters. The first is the opaque data pointer, the second is the Id that was assigned to the file while scanning. The fourth parameter is an integer. If it is non-zero, then the function is supposed to retrieve the file in question, store it on local disk, and write the resulting filename into the area to which the third parameter (a `char*` pointer) points. A fourth parameter of zero indicates that the decoder is done handling the file, so that the function can decide whether or not to remove the file.

The function must return `UURET_OK` upon success, or any other appropriate error code upon failure.

Since it can usually be assumed that disk space is plentily available, and storing a file is "cheaper" than retrieving it twice, this mechanism has not been used so far.

3.5 Filename Filter

For portability reasons, the library does not make any assumptions of the legality of certain filenames. It will pick up a "garbage" file name from the encoded file and happily use it if not told otherwise. For example, on DOS systems many filenames must be truncated in order to be valid.

If a "Filename Filter" is installed, the library will pass each potential filename to the filter and then use the filename that the filter function returns. The filter also has to remove all directory information from the filename – the library itself does not know about directories at all.

The filter function receives the potential filename as string and must return a pointer to a string with the corrected filename. It may either return a pointer to some position in the original string or a pointer to some static area, but it should not modify the source string.

Two examples of filename filters can be found among the UUDeview distribution as `uufnflt.c`. The DOS filter function disposes directory information, uses only the first 8 characters of the base filename and the first three characters after the last '.' (since a filename might have two extensions). Also, space

[4] This happens if, in a MIME multipart posting, the final boundary cannot be found. After searching the boundary until the end-of-file, the scanner resets itself to the location of the previous boundary.

characters are replaced by underscores. The Unix filter just returns a pointer to the filename part of the name (without directory information).

The "garbage" filename mentioned above was just for the sake of argument. It is generally safe to assume that the input filename is not too weird; after all, it is a filename valid on *some* system. Still, the user should always be granted the possibility of renaming a file before decoding it, to allow decoding of files with insane filenames.

4 The File List

While scanning the input files, a linked list is built. Each node is of type `uulist` and describes one file, possibly composed of several parts. This section describes the members of the structure that may be of interest to an application.

`state` Describes the state of this file. Either the value `UUFILE_READ`[5] or a bitfield of the following values:

- **`UUFILE_MISPART`** The file is missing at least one part. This bit is set if the part numbers are non-sequential. Usually results in incorrect decoding.
- **`UUFILE_NOBEGIN`** No "begin" line was detected. Since *Base64* files do not have begin lines, this bit is never set on them. For *BinHex* files, the initial colon is used.
- **`UUFILE_NOEND`** No "end" line was detected. Since *Base64* files do not have end lines, this bit is never set on them. A missing end on *uuencoded* or *xxencoded* files usually means that the file is incomplete. For *BinHex*, the trailing colon is used as end marker.
- **`UUFILE_NODATA`** No encoded data was found within these parts.
- **`UUFILE_OK`** This file appears to be okay, and decoding is likely to be successful.
- **`UUFILE_ERROR`** A decode operation was attempted, but failed, usually because of an I/O error.
- **`UUFILE_DECODED`** This file has already been successfully decoded.
- **`UUFILE_TMPFILE`** The file has been decoded into a temporary file, which can be found using the `binfile` member (see below). This flag gets removed if the temporary file is deleted.

`mode` For *uuencoded* and *xxencoded* files, this is the file mode found on the "begin" line, *Base64* and *BinHex* files receive a default of 0644. A decode operation will try to restore this mode.

`uudet` The type of encoding this file uses. May be 0 if `UUFILE_NODATA` or one of the following values:

- **`UU_ENCODED`** for *uuencoded* data,
- **`B64ENCODED`** for *Base64* encoded data,
- **`XX_ENCODED`** for *xxencoded* data,
- **`BH_ENCODED`** for *BinHex* data,
- **`PT_ENCODED`** for plain-text "data", or
- **`QT_ENCODED`** for MIME *quoted-printable* encoded text.

`size` The approximate size of the resulting file. It is an estimated value and can be a few percent off the final value, hence the suggestion to display the size in kilobytes only.

`filename` The filename. For *uuencoded* and *xxencoded* files, it is extracted from the "begin" line. The name of *BinHex* files is encoded in the first data bytes. *Base64* files have the filename given in the "Content-Type" header. This field may be `NULL` if `state!=UUFILE_OK`.

[5] This value should only appear internally, never to be seen by an application.

subfname A unique identifier for this group of parts, usually derived from the "Subject" header of each part. It is possible that two nodes with the same identifier exist in the file list: If a group of files is considered "complete", a new node is opened up for more parts with the same Id.

mimeid Stores the "id" field from the "Content-Type" information if available. Actually, this Id is the first choice for grouping of files, but not surprisingly, non-MIME mails or articles do not have this information.

mimetype Stores this part's "Content-Type" if available.

binfile After decoding, this is the name of the temporary file the data was decoded to and stored in. This value is non-NULL if the flag `UUFILE_TMPFILE` is set in the state member above.

haveparts The part numbers found for this group of files as a zero-terminated ordered integer array. Some extra care must be taken, because a file may have a zeroth part as its first part. Thus if `haveparts[0]` is zero, it indicates a zeroth part, and the list of parts continues. A file may have at most one zeroth part, so if both `haveparts[0]` and `haveparts[1]` are zero, the zeroth part is the only part of this file.

No more than 256 parts are listed here.

misparts Similar to `haveparts`; a zero-terminated ordered integer array of missing parts, or simply `NULL` if no parts are missing. Since we don't mind if a file doesn't have a zeroth part, this array does not have the above problems.

5 Return Values

Most of the library functions return a value indicating success or the type of error occurred. The following values can be returned:

UURET_OK The action completed successfully.

UURET_IOERR An I/O error occurred. There may be many reasons from "File not found" to "Disk full". This return code indicates that the application should consult `errno` for more information.

UURET_NOMEM A `malloc()` operation returned `NULL`, indicating that memory resources are exhausted. Never seen this one in a VM system.

UURET_ILLVAL You tried to call some operation with invalid parameters.

UURET_NODATA An attempt was made to decode a file, but no encoded data was found within its parts. Also returned if decoding a *uuencoded* or *xxencoded* file with missing "begin" line.

UURET_NOEND A decoding operation was attempted, but the decoded data didn't have a proper "end" line. A similar condition can also be detected for *BinHex* files (where the colon is used as end marker).

UURET_UNSUP You tried to encode using an unsupported communications channel, for example piping to a command on a system without pipes.

UURET_EXISTS The target file already exists (upon decoding), and you didn't allow to overwrite existing files.

UURET_CONT This is a special return code, indicating that the current operation must be continued. This return value is used only by two encoding functions, so see the documentation there.

UURET_CANCEL The current operation was canceled, meaning that the Busy Callback returned a non-zero value usually because of user request. The library does not produce this return value on its own, so if your Busy Callback always returns zero, there's no need to handle this "Error".

6 Options

An application program can set and query a number of options. Some of them are read-only, but others can modify the behavior quite drastically. Some of them are intended to be set by the end user via an options menu.

UUOPT_VERSION (string, read-only)
: Retrieves the full version number of the library, composed as *MAJOR.MINOR*pl*PATCH* (the major and minor version numbers and the patchlevel are integers).

UUOPT_FAST (integer, default=0)
: If set to 1, the library will assume that each input file consists of exactly one email message or newsgroup posting. After finding encoded data within a file, the scanner will not continue to look for more data below. This strategy can save a lot of time, but has the drawback that files also cannot be checked for completeness – since the scanner does not look for "end" lines, we don't notice them missing.

  This flag does not have any effect on MIME multipart messages, which are always scanned to the end (alas, the Epilogue will be skipped). Actually, with this flag set, the scanner becomes more MIME-compliant.

UUOPT_DUMBNESS (integer, default=0)
: As already mentioned, the library evaluates information found in the part's "Subject" header line if available. The heuristics here are versatile, but cannot be guaranteed to be completely failure-proof. If false information is derived, the parts will be ordered and grouped wrong, resulting in wrong decoding.

  If the "dumbness" is set to 1, the code to derive a part number is disabled; it will then be assumed that all parts within a group appear in correct order: the first one is assigned number 1 etc. However, part numbers found in MIME-headers are still used (I haven't yet found a file where these were wrong).

  A dumbness of 2 also switches off the code to select a unique identifier from the subject line. This does still work with single-part files[6] and *might* work with multi-part files, as long as they're in correct order and not mixed. The filename is found on the first part and then passed on to the following parts.

  This option only takes effect for files scanned afterwards.

UUOPT_BRACKPOL (integer, default=0)
: Series of multi-part postings on the Usenet usually have subject lines like "You must see this! [1/3] (2/4)". How to parse this information? Is this the second part of four in a series of three postings, or is it the first of three parts and the second in a series of four postings? The library cannot know, and simply gives numbers in () parentheses precedence over number in [] brackets. If this assumption fails, the parts will be grouped and ordered completely wrong.

  Setting the "bracket policy" to 1 changes this precedence. If now both parentheses and brackets are present, the numbers within brackets will be evaluated first.

  This option only takes effect for files scanned afterwards.

UUOPT_VERBOSE (integer, default=1)
: If set to 0, the Message Callback will not be bothered with messages of level `UUMSG_MESSAGE` or `UUMSG_NOTE`. The default is to generate these messages.

UUOPT_DESPERATE (integer, default=0)
: By default, the library refuses to decode incomplete files and generates errors. But if switched into "desperate mode" these kinds of errors are ignored, and all *available* data is decoded. The usefulness of the resulting corrupt file depends on the type of the file.

[6] Of course, this option wouldn't make sense with single-part files, since there's no "grouping" involved that might fail.

UUOPT_IGNREPLY (integer, default=0)

If set to 1, the library will ignore email messages and news postings which were sent as "Reply", since they are less likely to feature useful data. There's no real reason to turn on this option any more (earlier versions got easily confused by replies).

UUOPT_OVERWRITE (integer, default=1)

When the decoder finds that the target file already exists, it is simply overwritten silently by default. If this option is set to 0, the decoder fails instead, generating a `UURET_EXIST` error.

UUOPT_SAVEPATH (string, default=(empty))

Without setting this option, files are decoded to the current directory. This "save path" is handled as prefix to each filename. Because the library does not know about directory layouts, the resulting filename is simply the concatenation of the save path and the target file, meaning that the path must include a final directory separator (slash, backslash, or whatever).

UUOPT_IGNMODE (integer, default=0)

Usually, the decoder tries to restore the file mode found on the "begin" line of *uuencoded* and *xxencoded* files. This is turned off if this option is set to 1.

UUOPT_DEBUG (integer, default=0)

If set to 1, all messages will be prefixed with the exact sourcecode location (filename and line number) where they were created. Might be useful if this is not clear from context.

UUOPT_ERRNO (integer, read-only)

This "option" can be queried after an operation failed with `UURET_IOERR` and returns the `errno` value that originally caused the problem. The "real" value of this variable might already be obscured by secondary problems.

UUOPT_PROGRESS (uuprogress, read-only)

Returns the progress structure. This would only make sense in multi-threaded environments where the decoder runs in one thread and is controlled from another. Although some care is taken while updating the structure's values, there might still be synchronization problems.

UUOPT_USETEXT (integer, default=0)

If this flag is true, plain text files will be presented for "decoding". This includes non-decodeable messages as well as plain-text parts from MIME multipart messages. Since they usually don't have associated filenames, a unique name will be created from a sequential four-digit number.

UUOPT_PREAMB (integer, default=0)

Whether to use the plain-text preamble and epilogue from MIME multipart messages. The standard defines they're supposed to be ignored, so there's no real reason to set this option.

UUOPT_TINYB64 (integer, default=0)

Support for tiny Base64 data. If set to off, the scanner does not recognize stand-alone Base64 encoded data with less than 3 lines. The problem is that in some cases plain text might be misinterpreted as Base64 data, since, for example, any four-character alphanumerical string like "Argh" appearing on a line of its own is valid Base64 data. Since encoded files are usually longer, and there is considerable confusion about erroneous Base64 detection, this option is off by default. There's probably no need to present this option separately to the user. It's reasonable to associate it with the "desperate mode" described above.

Note that this option only affects *stand-alone* data. Input from Mime messages with the encoding type correctly specified in the "Content-Transfer-Encoding" header is always evaluated.

There is also no problem with encoding types different than Base64, since they have an explicit notion of the beginning and end of a file, and no danger of misinterpretation exists.

UUOPT_ENCEXT (string, default=(empty))

When encoding into a file on the local disk, the target files usually receive an extension composed of the three-digit part number. This may be considered inappropriate for single-part files. If this option is set, its value is attached to the base file name as extension for the target file. A dot '.' is inserted automatically. When using uuencoding, a sensible value might be "uue".

This option does not alter the behaviour on multi-part files, where the individual parts always receive the three-digit part number as extension.

UUOPT_REMOVE (integer, default=0)

If true, input files are deleted if data was successfully decoded from them. Be careful with this option, as the library does not care if the file contains any other useful information besides the decoded data. And it also does not and can not check the integrity of the decoded file. Therefore, if in doubt of the incoming data, you should do a confidence check first and then delete the relevant input files afterwards. But then, this option was requested by many users.

UUOPT_MOREMIME (integer, default=0)

Makes the library behave more MIME-compliant. Normally, some liberties are taken with seemingly MIME files in order to find encoded data within it, therefore also finding files within broken MIME messages. If this option is set to 1, the library is more strict in its handling of MIME files, and will for example not allow Base 64 data outside of properly tagged subparts, and will not accept "random" encoded data.

You can also set the value of this option to 2 to enforce strict MIME adherance. If the option is 1, the library will still look into plain text attachments and try to find encoded data within it. This causes for example uuencoded files that were then sent in a MIME envelope to be recognized. With an option value of 2, the library won't even do that, trusting all MIME header information.

7 General Functions

After describing all the framework in the previous chapters, it is time to mention some function calls. Still, the functions presented here don't actually *do* anything, they just query and modify the behavior of the core functions.

int UUInitialize (void)

This function initializes the library and must be called before any other decoding or encoding function. During initialization, several arrays are allocated. If memory is exhausted, `UURET_NOMEM` is returned, otherwise the initialization will return successfully with `UURET_OK`.

int UUCleanUp (void)

Cleans up all resources that have been allocated during a program run: memory structures, temporary files and everything. No library function may be called afterwards, with the exception of `UUInitialize` to start another run.

int UUGetOption (int opt, int *ival, char *cval, int len)

Retrieves the configuration option (see section 6) opt. If the option is integer, it is stored in `ival` (only if `ival!=NULL`) and also returned as return value. String options are copied to `cval`. Including the final nullbyte, at most `len` characters are written to `cval`. If the progress information is queried with `UUOPT_PROGRESS`, `cval` must point to a `uuprogress` structure and `len` must equal `sizeof(uuprogress)`.

For integer options, `cval` may be NULL and `len` 0 and vice versa: for string options, `ival` is not evaluated.

int UUSetOption (int opt, int ival, char *cval)

Sets one of the configuration options. Integer options are set via `ival` (`cval` may be `NULL`), and string options are copied from the null-terminated string `cval` (`ival` may be 0). Returns `UURET_ILLVAL` if you try to set a read-only value, or `UURET_OK` otherwise.

char *UUstrerror (int errcode)

Maps the return values `UURET_*` into error messages:

UURET_OK "OK"

UURET_IOERR "File I/O Error"

UURET_NOMEM "Not Enough Memory"

UURET_ILLVAL "Illegal Value"

UURET_NODATA "No Data found"

UURET_NOEND "Unexpected End of File"

UURET_UNSUP "Unsupported function"

UURET_EXISTS "File exists"

int UUSetMsgCallback (void *opaque, void (*func) ())

Sets the Message Callback function to `func` (see section 3.2). `opaque` is the opaque data pointer that is passed untouched to the callback whenever it is called. To prevent compiler warnings, a prototype of the callback should appear before this line. Always returns `UURET_OK`. If `func==NULL`, the callback is disabled.

int UUSetBusyCallback (void *, void (*func) (), long msecs)

Sets the Busy Callback function to `func` (see section 3.3). `msecs` gives a timespan in milliseconds; the library will try to call the callback after this timespan has passed. On some systems, the time can only be queried with second resolution – in that case, timing will be quite inaccurate. The semantics for the other two parameters are the same as in the previous function. If `func==NULL`, the busy callback is disabled.

int UUSetFileCallback (void *opaque, int (*func) ())

Sets the File Callback function to `func` (see section 3.4). Semantics identical to the previous two functions. There is no need to install a file callback if this feature isn't used.

int UUSetFNameFilter (void *opaque, char * (*func) ())

Sets the Filename Filter function to `func` (see section 3.5). Semantics identical to the previous three functions. If no filename filter is installed, any filename is accepted. This may result in failures to write a file because of an invalid filename.

char * UUFNameFilter (char *fname)

Calls the current filename filter on `fname`. This function is provided so that certain parts of applications do not need to know which filter is currently installed. This is handy for applications that are supposed to run on more than one system. If no filename filter is installed, the string itself is returned. Since a filename filter may return a pointer to static memory or a pointer into the parameter, the result from this function must not be written to.

8 Decoding Functions

Now for the more useful functions. The functions within this section are everything you need to scan and decode files.

int UULoadFile (char *fname, char *id, int delflag)

Scans a file for encoded data and inserts the result into the file list. Each input file must only be scanned once; it may contain many parts as well as multiple encoded files, thus it is possible that many decodeable files are found after scanning one input file. On the other hand it is also possible that *no* decodeable data is found. There is no limit to the number of files.[7]

[7] Strictly speaking, the memory is of course limited. But try to fill a sensible amount with structures in the 100-byte region.

If id is non-NULL, its value is used instead of the filename, and the file callback is used to map the id back into a filename whenever this input file is needed again. If id *is* NULL, then the input file must not be deleted or modified until UUCleanUp has been called.

If delflag is non-zero, the input file will automatically be removed within UUCleanUp. This is useful when the decoder's input are also temporary files – this way, the application can forget about them right after they're "loaded". The value of delflag is ignored, however, if id is non-NULL; combining both options does not make sense.

The behavior of this function is influenced by some of the options, most notably UUOPT_FAST. The two most probable return values are UURET_OK, indicating successful completion, or UURET_IOERR in case of some error while reading the file. The other return values are less likely to appear.

Note that files are even scheduled for destruction if an error *did* happen during scanning (with the exception of a file that could not be opened). But error handling is slightly problematic here anyway, since it might be possible that useful data was found before the error occurred.

int UULoadFileWithPartNo (char *fname, char *id, int delflag, int partno)

Same as above, but assigns a part number to the data in the file. This function can be used if the callee is certain of the part number and there is thus no need to depend on UUDeview's heuristics. However, it must not be used if the referenced file may contain more than one piece of encoded data.

uulist * UUGetFileListItem (int num)

Returns a pointer to the numth item of the file list. The elements of this structure are described in section 4. The list is zero-based. If num is out-of-range, NULL is returned. Usage of this function is pretty straightforward: loop with an increasing value until NULL is returned. The structure must not be modified by the application itself. Also, none of the structure's value should be "cached" elsewhere, as they are not constant: they may change after each loaded file.

int UURenameFile (uulist *item, char *newname)

Renames one item of the file list. The old name is discarded and replaced by newname. The new name is copied and may thus point to volatile memory. The name should be a local filename without any directory information, which would be stripped by the filename filter anyway.

int UUDecodeToTemp (uulist *item)

Decodes the given item of the file list and places the decoded output into a temporary file. This is intended to allow "previews" of an encoded file without copying it to its final location (which would probably overwrite other files). The name of the temporary file can be retrieved afterwards by re-retrieving the node of the file list and looking at its binfile member.

UURET_OK is returned upon successful completion. Most other error codes can occur, too. UURET_NODATA is returned if you try to decode parts without encoded data or with a missing beginning (*uuencoded* and *xxencoded* files only) – of course, this condition would also have been obvious from the state value of the file list structure.

The setting of UUOPT_DESPERATE changes the behavior if an unexpected end of file was found (usually meaning that one or more parts are missing). Normally, the partially-written target file is removed and the value UURET_NOEND is returned. In desperate mode, the same error code is returned, but the target file is not removed.

The target file is removed in all other error conditions.

int UURemoveTemp (uulist *item)

After a file has been decoded into a temporary file and is needed no longer, this function can be called to free the disk space immediately instead of having to wait until UUCleanUp. If a decode operation is called for later on, the file will simply be recreated.

int UUDecodeFile (uulist *item, char *target)

This is the function you have been waiting for. The file is decoded and copied to its final location.

Calling `UUDecodeToTemp` beforehand is not required. If `target` is non-NULL, then it is immediately used as filename for the target file (without prepending the save path and without passing it through the filename filter). Otherwise, if `target==NULL`, the final filename is composed by concatenating the save path and the result of the filename filter used upon the filename found in the encoded file.

If the target file already exists, the value of the `UUOPT_OVERWRITE` option is checked. If it is false (zero), then the error `UURET_EXISTS` is generated and decoding fails. If the option is true, the target file is silently overwritten.[8]

The file is first decoded into a temporary file, then the temporary file is copied to the final location. This is done to prevent overwriting target files with data that turns out too late to be invalid.

`int UUInfoFile (uulist *item, void *opaque, int (*func) ())`

This function can be used to query information about the encoded file. This is either the zeroth part of a file if available, or the beginning of the first part up to the encoded data otherwise. Once again, a callback function is used to do the job. `func` must be a function with two parameters. The first one is an opaque data pointer (the value of `opaque`), the other is one line of info about the file (at maximum, 512 bytes). The callback is called for each line of info.

The callback can return either zero, meaning that it can accept more data, or non-zero, which immediately stops retrieval of more information.

Usually, the opaque pointer holds some information about a text window, so that the callback knows where to print the next line. In a terminal-oriented application, the user can be queried each 25th line and the callback can return non-zero if the user doesn't wish to continue.

`int UUSmerge (int pass)`

Attempts a "Smart Merge" of parts that seem to belong to different files but which *could* belong to the same. Occasionally, you will find a posting with parts 1 to 3 and 5 to 8 of "picture.gif" and part 4 of "picure.gif" (note the spelling). To the human, it is obvious that these parts belong together, to a machine, it is not. This function attempts to detect these conditions and merge the appropriate parts together. This function must be called repeatedly with increasing values for "pass": With `pass==0`, only immediate fits are merged, increasing values allow greater "distances" between part numbers,

This function is a bunch of heuristics, and I don't really trust them. In some cases, the "smart" merge may do more harm than good. This function should only be called as last resort on explicit user request. The first call should be made with `pass==0`, then with `pass==1` and at last with `pass=99`.

9 Encoding Functions

There are a couple of functions to encode data into a file. You will usually need no more than one of them, depending on the job you want to do. The functions also differ in the headers they generate. Some functions do generate full MIME-compliant headers. This may sound like the best choice, but it's not always the wisest choice. Please follow the following guidelines.

- Do not produce MIME-compliant messages if you cannot guarantee their proper handling. For example, if you create a MIME-compliant message on disk, and the user *includes* this file in a text message, the headers produced for the encoded data become not part of the final message's header but are just included in the message body. The resulting message will *not* be MIME-compliant!
- Take it from the author that slightly-different-than-MIME messages give the recipient much worse headaches than messages that do not try to be MIME in the first place.
- Because of that, headers should *only* be generated if the application itself handles the final mailing or posting of the message. Do not rely on user actions.

[8] If we don't have permission to overwrite the target file, an I/O error is generated.

- Do not encode to *Base64* outside of MIME messages. Because some information like the filename is only available in the MIME-message framework, *Base64* doesn't make much sense without it.
- However, if you can guarantee proper MIME handling, *Base64* should be favored above the other types of encoding. Most MIME-compliant applications do not know the other encoding types.

All of the functions have a bunch of parameters for greater flexibility. Don't be confused by their number, usually you'll need to fill only a few of them. There's a number of common parameters which can be explained separately:

`FILE *outfile`
The output stream, where the encoded data is written to.

`FILE *infile, char *infname`
Where the input data shall be read from. Only one of both values must be specified, the other can be NULL.

`char *outfname`
The name by which the recipient will receive the file. It is used on the "begin" line for *uuencoded* and *xxencoded* data, and in the headers of MIME-formatted messages. If this parameter is NULL, it defaults to `infname`. It must be specified if data is read from a stream and `infname==NULL`.

`int filemode`
For *uuencoded* and *xxencoded* data, the file permissions are encoded into the "begin" line. This mode can be specified here. If the value is 0, it will be determined by performing a `stat()` call on the input file. If this call should fail, a value of 0644 is used as default.

`int encoding`
The encoding to use. One of the three constants `UU_ENCODED`, `XX_ENCODED` or `B64ENCODED`.

Now for the functions ...

```
int UUEncodeMulti  (FILE *outfile, FILE *infile,
                    char *infname, int encoding,
                    char *outfname, char *mimetype,
                    int filemode)
```

Encodes data into a subpart of a MIME "multipart" message. Appropriate "Content-Type" headers are produced, followed by the encoded data. The application must provide the envelope and boundary lines. If `mimetype!=NULL`, it is used as value for the "Content-Type" field, otherwise, the extension from `outfname` or `infname` (if `outfname==NULL`) is used to look up the relevant type name.

```
int UUEncodePartial  (FILE *outfile, FILE *infile,
                      char *infname, int encoding,
                      char *outfname, char *mimetype,
                      int filemode, int partno,
                      long linperfile)
```

Encodes data as the body of a MIME "message/partial" message. This type allows message fragmentation. This function must be called repetitively until it runs out of input data. The application must provide a valid envelope with a "message/partial" content type and proper information about the part numbers.

Each call produces `linperfile` lines of encoded output. For *uuencoded* and *xxencoded* files, each output line encodes 45 bytes of input data, each *Base64* line encodes 57 bytes. If `linperfile==0`, this function is equivalent to `UUEncodeMulti`.

Different handling is necessary when reading from an input stream (if `infile!=NULL`) compared to reading from a file (if `infname!=NULL`). In the first case, the function must be called until

feof() becomes true on the input file, or an error occurs. In the second case, the file will be opened internally. Instead of UURET_OK, a value of UURET_CONT is returned for all but the last part.

```
int UUEncodeToStream  (FILE *outfile, FILE *infile,
                       char *infname, int encoding,
                       char *outfname, int filemode)
```

Encodes the input data and sends the plain output without any headers to the output stream. Be aware that for *Base64*, the output does not include any information about the filename.

```
int UUEncodeToFile  (FILE *infile, char *infname,
                     int encoding, char *outfname,
                     char *diskname, long linperfile)
```

Encodes the input data and writes the output into one or more output files on the local disk. No headers are generated. If diskname==NULL, the names of the encoded files are generated by concatenating the save path (see the UUOPT_SAVEPATH option) and the base name of outfname or infname (if outfname==NULL).

If diskname!=NULL and does not contain directory information, the target filename is the concatenation of the save path and diskname. If diskname is an absolute path name, it is used itself.

From the so-generated target filename, the extension is stripped. For single-part output files, the extension set with the UUOPT_ENCEXT option is used. Otherwise, the three-digit part number is used as extension. If the destination file does already exist, the value of the UUOPT_OVERWRITE is checked; if overwriting is not allowed, encoding fails with UURET_EXISTS.

```
int UUE_PrepSingle  (FILE *outfile, FILE *infile,
                     char *infname, int encoding,
                     char *outfname, int filemode,
                     char *destination, char *from,
                     char *subject, int isemail)
```

Produces a complete MIME-formatted message including all necessary headers. The output from this function is usually fed directly into a mail delivery agent which honors headers (like "sendmail" or "inews").

If from!=NULL, it is sent as the sender's email address in the "From" header field. Some MDA programs are able to provide the sender's address themselves, so this value may be NULL in certain cases.

If subject!=NULL, the text is included in the "Subject" header field. The subject is extended with information about the file name and part number (in this case, always "(001/001)").

"Destination" must not be NULL. Depending on the "isemail" flag, its contents are sent either in the "To" or "Newsgroups" header field.

```
int UUE_PrepPartial  (FILE *outfile, FILE *infile,
                      char *infname, int encoding,
                      char *outfname, int filemode,
                      int partno, long linperfile,
                      long filesize,
                      char *destination, char *from,
                      char *subject, int isemail)
```

Similar to UUE_PrepSingle, but produces a complete MIME-formatted "message/partial" message including all necessary headers. The function must be called repetitively until it runs out of input data. For more explanations, see the description of the function UUEncodePartial above.

The only additional parameter is filesize. Usually, this value can be 0, as the size of the input file can usually be determined by performing a stat() call. However, this might not be possible if infile refers to a pipe. In that case, the value of filesize is used.

```
#include <stdio.h>
#include <stdlib.h>
#include "config.h"
#include "uudeview.h"

int main (int argc, char *argv[])
{
  UUInitialize ();
  UULoadFile   (argv[1], NULL, 0);
  UUDecodeFile (UUGetFileListItem (0), NULL);
  UUCleanUp    ();
  return 0;
}
```

Figure 2: The "Trivial Decoder", Version 1

If the size of the input data cannot be determined, and `filesize` is 0, the function refuses encoding into multiple files and produces only a single stream of output.

If data is read from a file instead from a stream (`infile==NULL`), the function opens the file internally and returns `UURET_CONT` instead of `UURET_OK` on successful completion for all but the last part.

10 The Trivial Decoder

In this section, we implement and discuss the "Trivial Decoder", which illustrates the use of the decoding functions. We start with the absolute minimum and then add more features and actually end up with a limited, but useful tool. For a full-scale frontend, look at the implementation of the "UUDeview" program. The sample code can be found among the documentation files as `td-v1.c`, `td-v2.c` and `td-v3.c`.

10.1 Version 1

The minimal decoding program is displayed in Figure 2. Only four code lines are needed for the implementation. `<stdlib.h>` defines `NULL`, `<uudeview.h>` declares the decoding library functions, and `<config.h>`, the library's configuration file, is needed for some configuration details[9].

After initialization, the file given as first command line parameter is scanned. No symbolic name is assigned to the file, so that we don't need a file callback. After the scanning, the encoded file is decoded and stored in the current directory by its native name.

Of course, there is much to complain about:

- No error checking is done. For example, does the input file exist?
- Only a single file can be scanned for encoded data.
- If more than one encoded file is found, only the first one is decoded, the others are ignored.
- No checking is done if there actually *is* encoded data in the file and whether this data is valid.

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>
#include "config.h"
#include "uudeview.h"

int main (int argc, char *argv[])
{
  uulist *item;
  int i, res;

  UUInitialize ();
  for (i=1; i<argc; i++)
    if ((res = UULoadFile (argv[i], NULL, 0)) != UURET_OK)
      fprintf (stderr, "could not load %s: %s\n",
               argv[i], (res==UURET_IOERR) ?
               strerror (UUGetOption (UUOPT_ERRNO, NULL, NULL, 0)) :
               UUstrerror(res));

  for (i=0; (item=UUGetFileListItem(i)) != NULL; i++) {
    if ((item->state & UUFILE_OK) == 0)
      continue;
    if ((res = UUDecodeFile (item, NULL)) != UURET_OK) {
      fprintf (stderr, "error decoding %s: %s\n",
               (item->filename==NULL)?"oops":item->filename,
               (res==UURET_IOERR) ?
               strerror (UUGetOption (UUOPT_ERRNO, NULL, NULL, 0)) :
               UUstrerror(res));
    }
    else {
      printf ("successfully decoded '%s'\n", item->filename);
    }
  }
  UUCleanUp ();
  return 0;
}
```

Figure 3: The "Trivial Decoder", Version 2

10.2 Version 2

The second version, printed in figure 3, addresses all of the above problems. The code size more than tripled, but that's largely because of the error messages.

All files given on the command line are scanned[10], and all encoded files are decoded. Of course, it is now also possible for an encoded file to span its parts over more than one input file. Appropriate error messages are printed upon failure of any step, and a success message is printed for successfully decoded files.

Apart from the program's unfriendliness that there is no user-interaction like selective decoding of files, choice of a target directory etc., there are only three more items to complain about:

- Errors and other messages produced within the library aren't displayed because there's no message callback.
- No filename filter is installed, so decoding of files with invalid filenames will fail; this especially includes filenames with directory information.
- No information is printed for invalid encoded files, or files with missing parts (they're simply skipped).

10.3 Version 3

This last section adds a simple filename filter (targeting at a DOS system with 8.3 filenames) and a simple message callback, which just dumps messages to the console. Figure 4 lists the changes with respect to version 2 (for the full listing, refer to the source file on disk).

The message callback, a one-liner, couldn't be simpler. The filename filter will probably not win an award for good programming style, but it does its job of stripping Unix-style or DOS-style directory names and using only the first 8 characters of the base filename and the first three characters of the extension. If the filename contains space characters, they're replaced by underscores. Note that `dname`, the storage for the resulting filename, is declared static, as it must be accessible after the filter function has returned.

For portability, the filename filter uses a replacement function from the `fptools` library instead of relying of a native implementation of the `strrchr` function.

Both callbacks are installed right after initializing the library. Since now the filename of the decoded file may be different from the filename of the file list structure, we recreate the resulting filename by calling the filename filter ourselves for display, so that the user knows where to look for the file.

11 Replacement functions

This section is a short reference for the replacement functions from the `fptools` library. Some of them may be useful in the application code as well. Most of these functions are pretty standard in modern systems, but there's also a few from the author's imagination. Each of the functions is tagged with information why this replacement exists:

- "nonstandard" (ns): this function is available on some systems, but not on others. Functions with this tag could be safely replaced with a native implementation.
- "feature" (f): the replacement adds some functionality with respect to the "original".
- "author"(a): just a function the author considered useful.

`void _FP_free (void *)` (f)

ANSI C guarantees that `free()` can be safely called with a `NULL` argument, but some old systems dump core. This replacement just ignores a `NULL` pointer and passes anything else to the original `free()`.

[9] Actually, only the definition of `UUEXPORT` is needed. You could omit `<config.h>` and define this value elsewhere, for example in the project definitions.

[10] With Microsoft compilers on MS-DOS systems, don't forget to link with `setargv.obj` to properly handle wildcards

... right after the #includes

```
#include "fptools.h"

void MsgCallBack (void *opaque, char *msg, int level)
{
  fprintf (stderr, "%s\n", msg);
}

char * FNameFilter (void *opaque, char *fname)
{
  static char dname[13];
  char *p1, *p2;
  int i;

  if ((p1 = _FP_strrchr (fname, '/')) == NULL)
    p1 = fname;
  if ((p2 = _FP_strrchr (p1, '\\')) == NULL)
    p2 = p1;
  for (i=0, p1=dname; *p2 && *p2!='.' && i<8; i++)
    *p1++ = (*p2==' ')?(p2++,'_'):*p2++;
  while (*p2 && *p2 != '.') p2++;
  if ((*p1++ = *p2++) == '.')
    for (i=0; *p2 && *p2!='.' && i<3; i++)
      *p1++ = (*p2==' ')?(p2++,'_'):*p2++;
  *p1 = '\0';
  return dname;
}
```

... within `main()` *after* `UUInitialize`

```
  UUSetMsgCallback (NULL, MsgCallBack);
  UUSetFNameFilter (NULL, FNameFilter);
```

... replacing the main loop's else

```
    else {
      printf ("successfully decoded '%s' as '%s'\n",
              item->filename,
              UUFNameFilter (item->filename));
    }
```

Figure 4: Changes for Version 3

char *_FP_strdup (char *ptr) (ns)
Allocates new storage for the string `ptr` and copies the string including the final nullbyte to the new location (thus "duplicating" the string). Returns `NULL` if the `malloc()` call fails.

char *_FP_strncpy (char *dest, char *src, int count) (f)
Copies text from the `src` area to the `dest` area, until either a nullbyte has been copied or `count` bytes have been copied. Differs from the original in that if `src` is longer than `count` bytes, then only `count`-1 bytes are copied, and the destination area is properly terminated with a nullbyte.

void *_FP_memdup (void *ptr, int count) (a)
Allocates a new area of `count` bytes, which are then copied from the `ptr` area.

int _FP_stricmp (char *str1, char *str2) (ns)
Case-insensitive equivalent of `strcmp`.

int _FP_strnicmp (char *str1, char *str2, int count) (ns)
Case-insensitive equivalent of `strncmp`.

char *_FP_strrchr (char *string, int chr) (ns)
Similar to `strchr`, but returns a pointer to the last occurrence of the character `chr` in `string`.

char *_FP_strstr (char *str1, char *str2) (ns)
Returns a pointer to the first occurrence of `str2` in `str1` or `NULL` if the second string does not appear within the first.

char *_FP_strrstr (char *str1, char *str2) (ns)
Similar to `strstr`, but returns a pointer to the last occurrence of `str2` in `str1`.

char *_FP_stristr (char *str1, char *str2) (a)
Case-insensitive equivalent of `strstr`.

char *_FP_strirstr (char *str1, char *str2) (a)
Case-insensitive equivalent of `strrstr`.

char *_FP_stoupper (char *string) (a)
Converts all alphabetic characters in `string` to uppercase.

char *_FP_stolower (char *string) (a)
Converts all alphabetic characters in `string` to lowercase.

int _FP_strmatch (char *str, char *pat) (a)
Performs glob-style pattern matching. `pat` is a string containing regular characters and the two wildcards '?' (question mark) and '*'. The question mark matches any single character, the '*' matches any zero or more characters. If `str` is matched by `pat`, the function returns 1, otherwise 0.

char *_FP_fgets (char *buf, int max, FILE *file) (f)
Extends the standard `fgets()`; this replacement is able to handle line terminators from various systems. DOS text files have their lines terminated by CRLF, Unix files by LF only and Mac files by CR only. This function reads a line and replaces whatever line terminator present with a single LF.

char *_FP_strpbrk (char *str, char *accept) (ns)
Locates the first occurrence in the string `str` of any of the characters in `accept`.

char *_FP_strtok (char *str, char *del) (ns)
Considers the string `str` to be a sequence of tokens separated by one or more of the delimiter characters given in `del`. Upon first call with `str!=NULL`, returns the first token. Later calls with `str==NULL` return the following tokens. Returns `NULL` if no more tokens are found.

char *_FP_cutdir (char *str) (a)
Returns the filename part of `str`, meaning everything after the last slash or backslash in the string. Now replaced with the concept of the filename filter.

char *_FP_strerror (int errcode) (ns)
A rather dumb replacement of the original one, which transforms error codes from `errno` into a human-readable error message. This function should *only* be used if no native implementation exists; it just returns a string with the numerical error number.

char *_FP_tempnam (char *dir, char *pfx) (ns)
The original is supposed to return a unique filename. The temporary file should be stored in `dir` and have a prefix of `pfx`. This replacement, too, should only be used if no native implementation exists. It just returns a temporary filename created by the standard `tmpnam()`, which not necessarily resides in a proper `TEMP` directory. The value returned by this function is an allocated memory area which must later be freed by calling `free`.

12 Known Problems

This section mentions a few known problems with the library, which the author considers to be "features" rather than "bugs", meaning that they probably won't be "fixed" in the near future.

- Encoding to *BinHex* is not yet supported.
- The checksums found in *BinHex* files are ignored.
- If both data and resource forks in a *BinHex* file are non-empty, the larger one is decoded. Non-Mac systems can only use one of them anyway (usually the "data" fork, the "resource" fork usually contains M68k or PPC machine code).

References

[RFC0822] Crocker, D., "Standard for the Format of ARPA Internet Text Messages", RFC 822, Network Working Group, August 1982.

[RFC1521] Borenstein, N., "MIME (Multipurpose Internet Mail Extensions) Part One", RFC 1521, Network Working Group, September 1993.

[RFC1741] Faltstrøm, P., Crocker, D. and Fair, E., "MIME Content Type for BinHex Encoded Files", RFC 1741, Network Working Group, December 1994.

[RFC1806] Troost, R., Dorner, S., "The Content-Disposition Header", RFC 1806, Network Working Group, June 1995.

RFC documents ("Request for Comments") can be downloaded from many ftp sites around the world.

Input Octet	1							
Input Bit	7	6	5	4	3	2	1	0
Output Data #1	5	4	3	2	1	0		
Output Data #2							5	4
Input Octet	2							
Input Bit	7	6	5	4	3	2	1	0
Output Data #2	3	2	1	0				
Output Data #3					5	4	3	2
Input Octet	3							
Input Bit	7	6	5	4	3	2	1	0
Output Data #3	1	0						
Output Data #4			5	4	3	2	1	0

Table 1: Bit mapping for Three-in-Four encoding

A Encoding Formats

The following sections describe the four most widely used formats for encoding binary data into plain text, *uuencoding*, *xxencoding*, *Base64* and *BinHex*. Another section shortly mentions *Quoted-Printable* encoding.

Other formats exist, like *btoa* and *ship*, but they are not mentioned here. *btoa* is much less efficient than the others. *ship* is slightly more efficient and will probably be supported in future.

Uuencoding, xxencoding and Base 64 basically work the same. They are all "three in four" encodings, which means that they take three octets[11] from the input file and encode them into four characters.

Three bytes are 24 bits, and they are divided into 4 sections of 6 bits each. Table 1 describes in detail how the input bits are copied into the output data bits. 6 bits can have values from 0 to 63; each of the "three in four" encodings now uses a character table with 64 entries, where each possible value is mapped to a specific character.

The advantage of three in four encodings is their simplicity, as encoding and decoding can be done by mere bit shifting and two simple tables (one for encoding, mapping values to characters, and one for decoding, with the reverse mapping). The disadvantage is that the encoded data is 33% larger than the input (not counting line breaks and other information added to the encoded data).

The before-mentioned *ship* data is more effective; it is a so-called *Base 85* encoding. Base 85 encodings take four input bytes (32 bits) and encode them into five characters. Each of this characters encode a value from 0 to 84; five characters can therefore encode a value from 0 to $85^5 = 4437053125$, covering the complete 32 bit range. Base 85 encodings need more "complicated" math and a larger character table, but result in only 25% bigger encoded files.

In order to illustrate the encodings and present some actual data, we will present the following text encoded in each of the formats:

```
This is a test file for illustrating the various
encoding methods. Let's make this text longer than
57 bytes to wrap lines with Base64 data, too.
Greetings, Frank Pilhofer
```

A.1 Uuencoding

A document actually describing uuencoding as a standard does not seem to exist. This is probably the reason why there are so many broken encoders and decoders around that each take their liberties with the

[11] The term "octet" is used here instead of "byte", since it more accurately reflects the 8-bit nature of what we usually call a "byte"

Data Value	+0	+1	+2	+3	+4	+5	+6	+7
0	‘	!	”	#	$	%	&	’
8	(	)	*	+	,	-	.	/
16	0	1	2	3	4	5	6	7
24	8	9	:	;	<	=	>	?
32	@	A	B	C	D	E	F	G
40	H	I	J	K	L	M	N	O
48	P	Q	R	S	T	U	V	W
56	X	Y	Z	[	\	]	^	_

Table 2: Encoding Table for Uuencoding

definition.

The following text describe the pretty strict rules for uuencoding that are used in the UUEnview encoding engine. The UUDeview decoding engine is much more relaxed, according to the general rule that you should be strict in all that you generate, but liberal in the data that your receive.

Uuencoded data always starts with a `begin` line and continues until the `end` line. Encoded data starts on the line following the begin. Immediately before the `end` line, there must be a single *empty* line (see below).

`begin` *mode filename*
. . . encoded data . . .
“empty” line
`end`

A.1.1 The `begin` Line

The `begin` line starts with the word `begin` in the first column. It is followed, all on the same line, by the *mode* and the *filename*.

mode is a three- or four-digit octal number, describing the access permissions of the target file. This mode value is the same as used with the Unix `chmod` command and by the `open` system call. Each of the three digits is a binary or of the values 4 (read permission), 2 (write permission) and 1 (execute permission). The first digit gives the user’s permissions, the second one the permissions for the group the user is in, and the third digit describes everyone else’s permissions. On DOS or other systems with only a limited concept of file permissions, only the first digit should be evaluated. If the “2” bit is not set, the resulting file should be read-only, the “1” bit should be set for COM and EXE files. Common values are `644` or `755`.

filename is the name of the file. The name *should* be without any directory information.

A.1.2 Encoded Data

The basic version of uencoding simply uses the ASCII characters 32-95 for encoding the 64 values of a three in for encoding. An exception[12] is the value 0, which would normally map into the space character (ASCII 32). To prevent problems with mailers that strip space characters at the beginning or end of the line, character 96 “ ‘ ” is used instead. The encoding table is shown in table 2.

Each line of uuencoded data is prefixed, in the first column, with the encoded number of encoded octets on this line. The most common prefix that you’ll see is ‘M’. By looking up ‘M’ in table 2, we see that it represents the number 45. Therefore, this prefix means that the line contains 45 octets (which are encoded into 60 $(45/3 * 4)$ plain-text characters).

In uuencoding, each line has the same length, normally, the length (excluding the end of line character) is 61. Only the last line of encoded data may be shorter.

If the input data is not a multiple of three octets long, the last triple is filled up with (one or two) nulls. The decoder can determine the number of octets that are to go into the output file from the prefix.

[12] . . . that is not always respected by old encoders

A.1.3 The Empty Line

After the last line of data, there must be an *empty* line, which must be a valid encoded line containing no encoded data. This is achieved by having a line with the single character " ‘ " on it (which is the prefix that encodes the value of 0 octets).

A.1.4 The `end` Line

The encoded file is then ended with a line consisting of the word `end`.

A.1.5 Splitting Files

Uuencoding does not describe a mechanism for splitting a file into two or more messages for separate mailing or posting. Usually, the encoded file is simply split into parts of more or less equal line count[13]. Before the age of smart decoders, the recipient had to manually concatenate the parts and remove the headers in between, because the headers of mail messages *might* just be valid uuencoded data lines, thus potentially corrupting the data.

A.1.6 Variants of Uuencoding

There are many variations of the above rules which must be taken into account in a decoder program. Here are the most frequent:

- Many old encoders do not pay attention to the special rule of encoding the 0 value, and encode it into a space character instead of the " ‘ " character. This is not an "error," but rather a potential problem when mailing or posting the file.
- Some encoders add a 62nd character to each encoded line: sometimes a character looping from "a" to "z" over and over again. This technique could be used to detect missing lines, but confuses some decoders.
- If the length of the input file is not a multiple of three, some encoders omit the "unnecessary" characters at the end of the last data line.
- Sometimes, the "empty" data line at the end is omitted, and at other times, the line is just completely empty (without the " ‘ ").

There is also some confusion how to properly terminate a line. Most encoders simply use the convention of the local system (DOS encoders using CRLF, Unix encoders using LF, Mac encoders using CR), but with respect to the MIME standard, the encoding library uses CRLF on all systems. This causes a slight problem with some Unix decoders, which look for "end" followed directly by LF (as four characters in total). Such programs report "end not found", but nevertheless decode the file correctly.

A.1.7 Example

This is what our sample text looks like as uuencoded data:

```
begin 600 test.txt
M5&AI<R!I<R!A('1E<W0@9FEL92!F;W(@:6QL=7-T<F%T:6YG('1H92!V87)I
M;W5S"F5N8V]D:6YG(&UE=&AO9',N($QE="=S(&UA:V4@=&AI<R!T97AT(&QO
M;F=E<B!T:&%N"C4W(&)Y=&5S('1O('=R87`@;&EN97,@=VET:"!"87-E-C0@
E9&%T82P@=&]O+@I'<F5E=&EN9W,L($9R86YK(%!I;&AO9F5R"@``
`
end
```

[13] Of course, encoded files must be split on line boundaries instead of at a fixed byte count.

Data Value	+0	+1	+2	+3	+4	+5	+6	+7
0	+	-	0	1	2	3	4	5
8	6	7	8	9	A	B	C	D
16	E	F	G	H	I	J	K	L
24	M	N	O	P	Q	R	S	T
32	U	V	W	X	Y	Z	a	b
40	c	d	e	f	g	h	i	j
48	k	l	m	n	o	p	q	r
56	s	t	u	v	w	x	y	z

Table 3: Encoding Table for Xxencoding

Data Value	+0	+1	+2	+3	+4	+5	+6	+7
0	A	B	C	D	E	F	G	H
8	I	J	K	L	M	N	O	P
16	Q	R	S	T	U	V	W	X
24	Y	Z	a	b	c	d	e	f
32	g	h	i	j	k	l	m	n
40	o	p	q	r	s	t	u	v
48	w	x	y	z	0	1	2	3
56	4	5	6	7	8	9	+	/

Table 4: Encoding Table for Base64 Encoding

A.2 Xxencoding

The xxencoding method was conceived shortly after the initial use of uuencoding. The first implementations of uuencoding did not realize the potential problem of using the space character for encoding data. Before this mistake was workarounded with the special case, another author used a different charset for encoding, composed of characters available on any system.

Xxencoding is absolutely identical to uuencoding with the difference of using a different mapping of data values into printable characters (table 3). Instead of 'M', a normal-sized xxencoded line is prefixed by 'h' (note that 'h' encodes 45, just as 'M' in uuencoding). The empty data line at the end consists of a single '+' character. Our sample file looks like the following:

```
begin 600 test.txt
hJ4VdQm-dQm-V65FZQrEUNaZgNG-aPr6UOKlgRLBoQa3oOKtb65FcNG-qML7d
hPrJn0aJiMqxYOKtb64pZR4VjN5Ai62lZR0Rn64pVOqIUR4VdQm-oNLVo64lj
hPaRZQW-oO43i0XIr647tR4Jn65Fj65RmML+UP4ZiNLAURqZoO0-0MLBZBXEU
ZN43oMGkUR4xj9Ud5QaJZR4ZiNrAg62NmMKtf63-dP4VjNaJm0U++
+
end
```

A.3 Base64 encoding

Base 64 is part of the *MIME* (Multipurpose Internet Mail Extensions) standard, described in [RFC1521], section 5.2. Sometimes, it is incorrectly referred to as "MIME encoding"; however, the MIME documents specify much more than just how to encode binary data. It defines a complete framework for attachments within E-Mails. Being part of a widely accepted standard, *Base64* has the advantage of being the best-specified type of encoding.

The general concept of three-in-four encoding is the same as with the previous two types, just another new character table to represent the values needs to be introduced (table 4). Note that this table differs from the *xxencoding* table only in a single character ('/' versus '-'). If a line of encoding does not feature either character, it may be difficult to tell which encoding is used on the line.

Data Value	+0	+1	+2	+3	+4	+5	+6	+7
0	!	”	#	$	%	&	’	(
8	)	*	+	,	-	0	1	2
16	3	4	5	6	8	9	@	A
24	B	C	D	E	F	G	H	I
32	J	K	L	M	N	P	Q	R
40	S	T	U	V	X	Y	Z	[
48	‘	a	b	c	d	e	f	h
56	i	j	k	l	m	p	q	r

Table 5: Encoding Table for BinHex Encoding

The *Base64* encoding does not have “begin” and “end” lines; such a concept is not needed, because the framework of a *MIME* message defines the beginning and end of a part. The encoded data is defined to be a “stream” of characters, and the decoder is supposed to ignore any “illegal” characters in the stream (such as line breaks or other whitespace). Each line must be shorter than 80 characters and terminated with a CRLF sequence. No particular line length is enforced, but most implementations encode 57 octets into 76 encoded characters. Theoretically, a line might hold 79 characters, although this would violate the rule of thumb that the line length is a multiple of four (therefore encoding an integral number of octets).[14]

The end-of-file handling if the input data has not a multiple of three octets is slightly different in *Base64* encoding than it is in uuencoding. If one octet is left at the end of the input stream, the data is padded with 4 zero bits (giving a total of 12 bits) and encoded into two characters. After that, two equal signs ‘=’ are written to complete the four character sequence. If two octets are left, the data is padded with 2 zero bits (giving a total of 18 bits), and encoded into three characters, after which a single equal sign ‘=’ is written.

Here’s our sample file in *Base64*. Note that this text is *only* the encoded data. It is not a valid *MIME* message. Without the required framework, no proper *MIME* software will read it.

```
VGhpcyBpcyBhIHRlc3QgZmlsZSBmb3IgaWxsdXN0cmF0aW5nIHRoZSB2YXJpb3VzCmVuY29kaW5n
IG1ldGhvZHMuIExldCdzIG1ha2UgdGhpcyB0ZXh0IGxvbmdlciB0aGFuCjU3IGJ5dGVzIHRvIHdy
YXAgbGluZXMgd2l0aCBCYXNlNjQgZGF0YSwgdG9vLgpHcmVldGluZ3MsIEZyYW5rIFBpbGhvZmVy
Cg==
```

For a more elaborate documentation of *Base64* encoding and details of the *MIME* framework, I suggest reading [RFC1521].

The *MIME* standard also defines a way to split a message into multiple parts so that re-assembly of the parts on the remote end is easily possible. For details, see section 7.3.2, “The Message/Partial subtype” of the standard.

A.4 BinHex encoding

The *BinHex* encoding originates from the Macintosh environment, and it takes the special properties of a Macintosh file into account. There, a file has two parts or “forks”: the “resource” fork holds machine code, and the “data” fork holds arbitrary data. For files from other systems, the data fork is usually empty.

I have not found a “definitive” definition of the format. My knowledge is based on two descriptions I found, one from Yves Lempereur and another from Peter Lewis. A similar description can be found in [RFC1741].

A *BinHex* file is a stream of characters, beginning and ending with a colon ‘:’; intermediate line breaks are to be ignored by the decoder. Each line but the last should be exactly 64 characters in length. The last line may be shorter, and in a special case can also be 65 characters long. The trailing colon must not stand alone, so if the input data ends on an output line boundary, the colon is appended to this line as 65th character. Thus a *BinHex* begins with a colon in the first column and ends with a colon *not* in the first column.

[14] Yes, there *are* files violating this assumption.

Compressed Data							Uncompressed Data					
00	11	22	33	44	55	↦	00	11	22	33	44	55
11	22	90	04	33		↦	11	22	22	22	22	33
11	22	90	00	33	44	↦	11	22	90	33	44	
2B	90	00	90	04	55	↦	2B	90	90	90	90	55

Table 6: BinHex RLE decoding

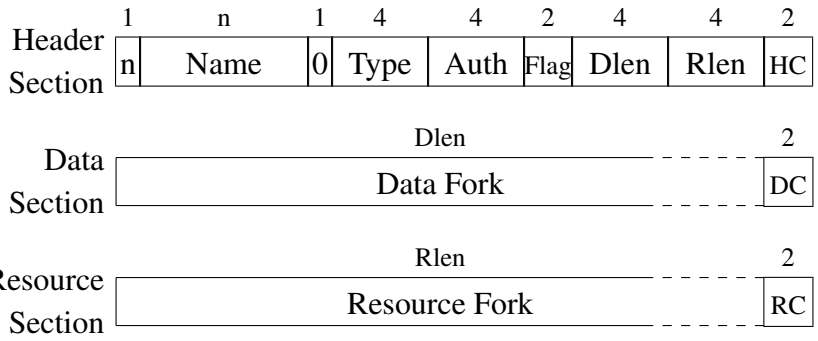


Figure 5: BinHex file structure

The line before the beginning of encoded data (before the initial ':') should contain the following verbatim text:[15]

```
(This file must be converted with BinHex 4.0)
```

BinHex is another three-in-four encoding, and not surprisingly, another different character table is used (table 5). The documentation does not explicitly mention what is supposed to happen if the original input data does not have a multiple of three octets. But from reading between the lines, it looks like "unnecessary" characters (those that would result in equal signs in Base64 encoding) are not printed.

The encoded characters decode into a RLE-compressed bytestream, which must be handled in the next step (of course, decoding and decompressing are usually handled at the same time). A Run Length Encoding simply replaces multiple subsequent occurrences of one octet are replaced by the character, a special marker, and the repetition count. BinHex uses the marker `0x90` (octal `0220`, decimal `128`). The octet sequence `0xff 0x90 0x04` would decompress into four times `0xff`. If the marker itself occurs, it must be "escaped" by the special sequence `0x90 0x00` (the marker with a repetition count of 0). Table 6 shows four more examples. Note the last example, where the marker itself is repeated.

The decompression results in a data stream which consists of three parts, the header section, the data fork and the resource fork. Figure 5 shows how the sections are composed. The numbers above each item indicate its size in octets. The header has the following items:

n The length of the filename in octets. This is a single octet, so the maximum length of a filename is 255.

Name The filename, *n* octets in length. The length does not include the final nullbyte (which is actually the next item).[16]

0 This single nullbyte terminates the previous filename.

Type The Macintosh file type.

Auth The Macintosh "creator", the program which wrote the original file. This and the previous item are used to start the right program to edit or display a file. I have no idea what common values are.

[15] In fact, this text is *required* by certain decoding software.

[16] The Filename may contain certain characters that are invalid on MS-DOS systems, like space characters

Flags Macintosh file flags. No idea what they are.

Dlen The number of octets in the data fork.

Rlen The number of octets in the resource fork.

HC CRC checksum of the header data.

After the header, at offset $n + 22$, follow the *Dlen* octets of the data fork and a CRC checksum of the data fork (offset $n + Dlen + 22$), then *Rlen* octets of the resource fork (offset $n + Dlen + 24$) and a CRC checksum of the resource fork (offset $n + Dlen + Rlen + 24$). Note that the CRCs are present even if the forks are empty.

The three CRC checksums are calculated as described in the following text, taken from Peter Lewis' description:

> BinHex 4.0 uses a 16-bit CRC with a 0x1021 seed. The general algorithm is to take data 1 bit at a time and process it through the following:
>
> 1. Take the old CRC (use 0x0000 if there is no previous CRC) and shift it to the left by 1.
> 2. Put the new data bit in the least significant position (right bit).
> 3. If the bit shifted out in (1) was a 1 then xor the CRC with 0x1021.
> 4. Loop back to (1) until all the data has been processed.

This is the sample file in *BinHex*. However, the encoder I used replaced the LF characters from the original file with CR characters. It probably noticed that the input file was plain text and reformatted it to Mac-style text, but I consider this a software bug. The assigned filename is "test.txt".

```
(This file must be converted with BinHex 4.0)
:#&4&8e3Z9&K8!&4&@&4dG(Kd!!!!!!#X!!!!!+3j9'KTFb"TFb"K)(4PFh3JCQP
XC5"QEh)JD@aXGA0dFQ&dD@jR)(4SC5"fBA*TEh9c$@9ZBfpND@jR)'ePG'K[C(-
Z)%aPG#Gc)'eKDf8JG'KTFb"dCAKd)'a[EQGPFL"dD'&Z$68h)'*jG'9c)(4[)(G
bBA!JE'PZCA-JGfPdD#"#BA0P0M3JC'&dB5`JG'p[,Je(FQ9PG'PZCh-X)%CbB@j
V)&"TE'K[CQ9b$B0A!!!!:
```

A.5 Quoted-Printable

The *Quoted-Printable* encoding is, like *Base64*, part of the *MIME* standard, described in [RFC1521]. It is not suitable for encoding arbitrary binary data, but is intended for "data that largely consists of octets that correspond to printable characters". It is widely in use in countries with an extended character set, where characters like the German umlauts 'ä' or 'ß' are represented by non-ASCII characters with the highest bit set.

The essence of the encoding is that arbitrary octets can be represented by an equal sign '=' followed by two hexadecimal digits. The equal sign itself, for example, is encoded as "=3D".

Quoted-Printable enforces a maximum line length of 76 characters. Longer lines can be wrapped using soft line breaks. If the last character of an encoded line is an equal sign, the following line break is to be ignored.

It would indeed be possible to transfer arbitrary binary data using this encoding, but care must be taken with line breaks, which are converted from native format on the sender's side and back into native format on the recipient's side. However, the native representations may differ. But this alternative is hardly worth considering, since for arbitrary data, *quoted-printable* is substantially less effective than *Base64*.

Please refer to the original document, [RFC1521], for a complete discussion of the encoding.

Here is how the example file could look like in Quoted-Printable encoding.

```
This is a test file for =
illustrating the various
encoding methods=2e=20=
Let=27s make this text=
```

```
 longer than
=357 bytes to wrap lines =
with Base64 data=2c too=2e
Greetings=2c Frank Pilhofer
```