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	V	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

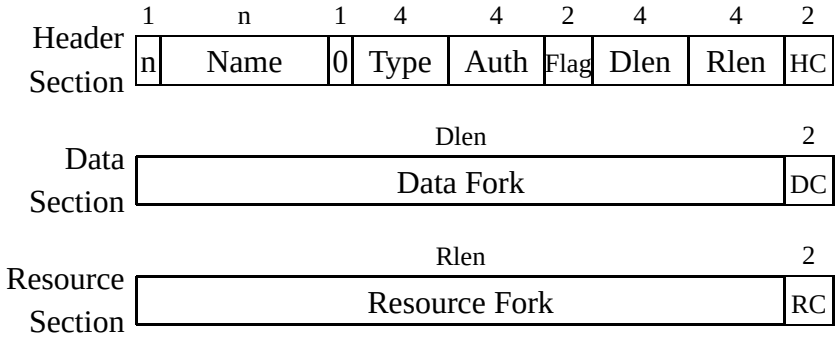

Figure 5: BinHex file structure

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 explicitely 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.

Flags Macintosh file flags. No idea what they 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

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
```