Explaining how UUEncoding works

Ready to Send

In this article, Steven Goodwin explains about the UUEncoding method. What it is, where it's used, and most importantly, how this transferring method works.

BY STEVEN GOODWIN

UUEncoding is a method to transmit binary files (such as MP3s, JPEGs, and executables) within a text-only medium – the most notable of which being email. It does this by encoding the data stream into a series of printable characters that can be included directly as if it were part of the text.

Although the format looks like unintelligible garbage, it does not provide any encryption facilities (see the "Eskimos and Egypt" boxout) and so should not be used for transmitting private or confidential data. It does, however, provide a very simple (and consequently, very quick) method for packaging files through email as an attachment, or part of an FTP to email gateway.

Although its primary purpose is for binary files, UUE can also work as an encoder for text files (perhaps as an alternative to rot13 for hiding spoiler information), but this case is less common. Partly because it buys you very little in terms of functionality, and partly because the text is treated exactly like binary data – meaning odd carriage return and line feed combinations that are present on some systems (cough!) will get replicated "as-is" on any machine to which the file is sent.

Back in Time

Unix-to-Unix Encoding (or UUE for short) was introduced in version 4.0 of BSD Unix, and is part of the same UU family that includes UUCP (Unix-to-Unix CoPy) and UUD (Unix-to-Unix decoding). Its usage is like any other command line tool:

```
uuencode fosdemtalk.mp3 ➥
talk_by_Steev.mp3 >fosdemtalk.uu
```

We need to specify some details, the first being the source filename that we want to encode. The second parameter is the name of the file that will be created when the UUE data is decoded, in this example it would be clear that the file has come from me.

Lastly, to prevent all our output finding its way to stdout, a redirection places the UU encoded text into a file. If no filenames are given, UUE will expect data from the standard input.

Instead of a redirection, you could, of course, pipe it directly to your mail client:

```
uuencode fun.jpg fun.jpg | mail➥
 -s "Check this" friend@work.com
```

Despite its age, UUE is still used today, most notable in Internet newsgroups. Pictures, sounds and programs are transmitted using uuencode, which has an unsurpassed dominance in the alt.binaries.* hierarchy.

Eskimos and Egypt

There is a very important difference between encoding and encryption. Encoding converts between one format and another for the purpose of transmission, storage, or access. Some formats may have a password attached to prevent it being decoded by 'unauthorised software' – such as the Microsoft Excel format.
However, as history has shown, it is a waste of time trying to protect file formats by encoding. Encoded formats will only stop some of the hackers, some of the time.
In contrast, the purpose of encryption is to stop all of the hackers, all of the time, thus providing a secure method of storing data which then may be transmitted and accessed only by those with the authority. The terms encoding and encryption should not be mixed up and used interchangeably.

Figure 1:A sample of a UU encoded file

```
begin 644 hello.txt
M5&AI<R!I<R!A('1E<WO@;V8@555%+B!)="=S(&YO="!B:6<L(&YO<B!I<R!I
M="!C;&5V97(L(&)U=`II="!A="!L96%S="!P<F]V97,@;7D@<&]I;G0A"@I)
M="!I<R!P<F]V:61E9"!A<R!A;B!E>&%M<&QE(&]F(&5N8V]D:6YG(&9O<B!T
M:&4*87)T:6-L92!I;B!,:6YU>"!-86=A>FEN92P@9F]R('=H:6-H('1H:7,@
:9FEL92!W87,*8W)E871E9"X*"E-T965V"@H`
`
end
```

As uuencode is a textual format, it is easy to chop the file into pieces, using standard Unix tools – such as *split*. The file can be reconstituted after it is received.

For example:

```
$ split -l 512 fosdemtalk.uu ➥
fosdemsplit_
```

will split the file fosdemtalk.uu into several smaller files, each 512 lines long. The files can be rejoined into one using the 'cat' command:

```
$ cat fosdemsplit_[a-z][a-z] ➥
>fosdemtalk.uu
```

This technique is especially useful if you are trying to bypass the size limit for attachments on a particular mail server. The only consideration to bear in mind is that you can not split the UUEd file in the middle of a line without great care – but that is only common sense.

I'm the Only One

The text given in Figure 1 is an example of a UU encoded file. As you can see, it consists of three basic parts: a beginning, a middle and an end.

The beginning section (which is helpfully named with the keyword, 'begin ') acts as a header and describes the file that follows. There are no white space characters before it, and only one (important) space afterwards. This separates 'begin ' from the three-digit number that indicates the *access mode* of the file. If you are not used to seeing the mode written as 644, but as a combination of the letters 'r', 'w', and 'x' (please see "All Right Now").

This access mode will be given to the newly created file, once it has been decoded. As you can see, UU *really* does stand for Unix-to-Unix – there's no Windows-specific perversions masquerading in here!

The filename is self-explanatory, and can include a path if you so wish. It is, along with the mode, the only part of the file you can safely change without risking corruption.

The middle section (with the letter M's down the left hand side) is the encoded section (which we will come to shortly), whilst the terminator is marked simply with 'end'. You will often see a back-tick (`) on the preceding line. This is actually part of the encoded data – not the 'end' tag. We shall see why later.

If you have split a large UUE file, do not include the 'begin' or 'end' lines on each section of the file. The presence of 'the Ms' is enough to indicate that this portion of the file needs to be re-joined to the whole (without any interspersed blank lines), to permit decoding. This should be done carefully, as UUE will produce a broken file if done incorrectly.

If a UUE file is split and transmitted through email it is usual to use the subject line to indicate to the user which part of file this is, as one UUEncoded data block looks much the same as all the rest!

```
Subject: fosdemtalk.mp3.uu 2/7
```

Each line of the encoded data may hold up to a maximum of 45 characters, and must end with a carriage return (and certainly not a CR/LF combination). Although some implementations support it, the standard Linux version does not, and usually results in a "No `end' line" error.

Bright Lights Big City

Every byte in a binary source file, be it an audio file or the code for some application, can be a value between 0 and 255. The 8-bit ASCII character does not have such a large range of characters, even though a single byte is used to represent each character used.

Any characters above value 127 are considered to be 'extended' characters, which cannot be relied on to work on all systems. See the 'Sign of the Times' box for the reason. This range is reduced by almost a quarter again because most of the characters with codes below 32 have special purposes like being used as terminal control codes (like the bell code) or are unusable because they are used for text formatting and layout, such as the carriage return and line feed characters.

With a simple piece of C code you will be able to see just how many characters

All Right Now

When shown on screen, the access mode for a file is given as a combination of the letters 'r', 'w', and 'x'.

```
-rw-r--r--    1 steev    users
206 Jan 30 09:41 hello.txt
```

As we know, there are three types of access group available under Linux: user, group, and everybody. Each of these groups have 8 possible combinations of attributes (see Table 1).

Base64

Modern implementations of UUE will also support encoding in base64. This does not have the trademark 'M's, and begins with the line "begin-base64". Its implementation differs and so shall not be covered here.

Table 1: File Attributes

No.	Access mode
0 =	—
1 =	--x
2 =	-w-
3 =	-wx
4 =	r--
5 =	r-x
6 =	rw-
7 =	rwx

So in our example, 644 indicates read/write access for the user, and read-only for the user's group, and everybody else. This is probably the most prominent application of octal (base 8) in Linux (for more information, see Sweet Sixteen).

really are available for use. A rather paltry 95! That is not enough to utilise 7 bits from each byte (as that would required 128 printable characters), but there is enough for 6-bit encoding (which only requires 64). So we take the first *6* bits from the first byte and represent it as a character – then we take the *2* left over bits and join to the first 4 of the next byte; we then take the *4* left over bits and join them onto the first *2* bits of the next byte; and finally we take the last *6* bits of this byte. This converts four bytes into three as seen in diagram 1.

This encoding method means that the new ASCII file will be about 33% larger than the original binary. The extra control characters (newlines and 'the Ms') will push this figure to around 35%.

For each 6-bit number we produce (which can range from 0 to 63) we need to offset (i.e. move) it into a range where all the (contiguous) values represent printable characters. The range chosen is between 32 and 95, so a simple add operation is all that's necessary. For those who can read 'C', we might demonstrate this with a code fragment similar to that in Listing 2.

Each mask is used to remove those bits that have either been encoded already, or those which map onto a different ASCII character. If you have trouble thinking of the masks, refer to the "Sweet Sixteen" boxout or think of them in binary.

There is however one special case which must be taken care of, and that is shown in Listing 3. Since character 32 is a space (and liable to be mis-interpreted), it is changed automatically by the encoder to the back-tick (`). This has a code of 96, and therefore is still part of our contiguous range, which now extends from 33 to 96.

Pop Musik

Now we understand the encoding method, we need to know how much data there is to handle. You will have noticed there is no 'file size' parameter anywhere in the format. It is not needed, since the start of each line begins with a single character (the 'M's we saw in the example), which indicate how many characters are present on the line, and can be anything from 0 to 45. Longer lines are not permitted because of UUE's heritage; that is, email systems might wrap them and corrupt the format as a consequence.

Encoding the 'number of characters on a line' is done in a similar way to the main encoding: the value to be encoded (which has already been restricted to 0 to 45, i.e. 6 bits) is offset by a specific

Trout Mask Replica

Masks are used to isolate specific bits within the encoding implementation. The example of 'char_3 = (byte_b << 2) & 0x3c' isolates bits 5,4,3 and 2. This maintains their value (be it 0 or 1), while clearing all other bits to zero.

The man pages for uuencode use 0x3f for this case. However, this will also mask bits 1 and 0, and allow them to be written into char_3. This, although not wrong (since these bits will be 0 anyway), should not be used from a programming standpoint because it implies we're interested in bits 1 and 0 from the (byte_b << 2) expression – which we're not! We only want 4 bits from it. Our mask should reflect that, so we use 0x3c.

Listing 1: Printable characters

```
int c, tot=0;

for(c=0;c<127;c++)
        if (isprint(c))
                tot++;
printf("There are %d printable characters!\n", tot);
```

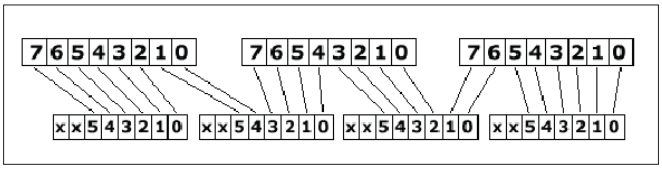

Diagram 1: Converting 4 bytes into 3

Sweet Sixteen

All of computer science is about numbers and what they represent. An eight bit byte, for example, is often spilt into two 4 bit numbers (between 0 and 15) which can be rendered with the hexadecimal number system (0-9, and the letters A to F). Those who grew up with the home computer era in the 1980s may remember machine code listings given as long strings of hexadecimal (aka hex or base 16) numbers.

Octal is similar inasmuch as each Octal digit (base 8) is a 3 bit number, whose value lies between 0 and 7. This means an 8-bit byte can be represented using two Octal digits, with 2 bits left over which may (depending on the circumstance) be ignored.

'C' directly supports both number bases because it often easier to think in hex or octal, than decimal.

```
int iOctalNumber = 0644;      /*
Value begins with a 0 means 644
is octal */
int iHexNumber = 0xC4;        /*
The 0x means C4 is hex -- the
case of 'C' is unimportant */
```

Sign of the Times

In order to represent negative numbers, computers use a system called 2's complement binary.

A byte, in itself, has no meaning. It's only when we say "this byte has a number in it", does it become meaningful. 2's complement binary says that when interpreting the meaning of this byte, each non-zero bit will correspond to a value, the most significant bit (MSB) of which will be negative:

	MSB						LSB
Bit:	7	6	5	4	3	2	1	0
Value:	-128	64	32	16	8	4	2	1

The range of a byte can therefore be from -128 to 127. It is a *signed* character.

If we interpret the byte as *un*signed (i.e. -128 is read as 128), the range becomes 0 to 255.

Because the founding language of Linux (and Unix) was 'C' – and 'C' doesn't specify whether its character type should be signed or unsigned, it can not be determined which range is valid for any specific machine. 128 may be a perfectly valid character code on an unsigned system, but break badly on a signed one. Therefore, only values between 0 and 127 can be guaranteed, and anything above 127 is considered 'extended'.

Listing 2: 6 bits to the byte code fragment

```
unsigned char byte_a, byte_b, byte_c;
unsigned char char_1, char_2, char_3, char_4;

        /* Retrieve byte_a, byte_b and byte_c from the data stream */
        byte_a = fgetc(fp);
        byte_b = fgetc(fp);
        byte_c = fgetc(fp);

        /* Map 3 bytes into 4 characters, each between 0 and 63 */
        /* see BOXOUT: Trout Mask Replica for more information */
        char_1  = (byte_a >> 2) & 0x3f;   /* First 6 bits of a */

        char_2  = (byte_a << 4) & 0x30; /* Last 2 bits of a */
        char_2 |= (byte_b >> 4) & 0x0f; /* First 4 bits of b */

        char_3  = (byte_b << 2) & 0x3c;   /* Last 4 bits of b */
        char_3 |= (byte_c >> 6) & 0x03; /* First 2 bits of c */

        char_4  = byte_c & 0x3f;        /* Last 6 bits of c */

        /* Offset to the 32-95 range */
        char_1 += 32;
        char_2 += 32;
        char_3 += 32;
        char_4 += 32;
```

number (also 32) so it can be represented as a printable character. With a usual line length of 45, this produces a value of 77: the ASCII character of which is the letter 'M'. It is present at the start of every line, except the last one.

When a line is of zero length we would normally produce character code 32 (i.e. the space). So, as before, a special case is made to represent it with a back-tick (`). This is why many UUE files appear to terminate with a back-tick *and* the keyword 'end'.

Paddy McCarthy

When the file size is not an exact multiple of 3, an extra one or two bytes of padding are added internally to make the algorithm work efficiently (which it does by removing the two 'special case' scenarios that could occur). The *de*coder will determine that the bytes are actually padding by noting that the line contains (say) 8 ASCII characters (implying a possible 6 bytes of data), but the first character of the line reports that there are (in fact) only 5 bytes of data to decode.

Listing 3: Dealing with Space

```
if (char_1 == 32)    char_1 = 96;
if (char_2 == 32)    char_2 = 96;
if (char_3 == 32)    char_3 = 96;
if (char_4 == 32)    char_4 = 96;
```

Another Night In

The decoding algorithm works as above, but (naturally enough) in reverse. The UUD program works simply as well:

```
uudecode textfile.uu
```

The program makes use of the 'begin ' line (remember that all important space!) to automatically strip off any email headers or other spurious junk that exists before it. Similarly, it will ignore anything after the 'end' line; as it should. The file created will (by default) have the same name and mode as indicated by the 'begin' line, overwriting any file that previously existed with that name – permissions permitting, of course.

You can ask uudecode to create the file with a different name (but same permissions) if you wish, by using the -o flag:

```
uudecode -o new_hello.txt ➥
textfile.uu
```

Or, instead of creating a file, it is possible to pipe the output into another program like this:

```
uudecode -o /dev/stdout ➥
textfile.uu | less
```

Living Next Door To Alice

Despite appearances and its apparent suitability, UUE is not used in MIME (Multi-Purpose Internet Mail Extensions) to send binary files as email attachments. This is because UUE can, as we have unfortunately seen, get corrupted.

However, where UUE scores highly, is that it is incredible easy to program, and can be handled by – and across – virtually every operating platform available.

Some mail clients will detect UUE data within the message, present it as an attachment, and then remove it from the body of the mail automatically.

It is not unknown for even the simplest of systems to provide such decoding functionality straight out of the box.

Its behaviour can also be problematic for files that are *very* big. And by very I mean those in excess of 2 gigabytes – although transferring such files should probably be left to FTP or DVD-R and snail mail!

Despite these issues, however, UUEncoding is a useful little tool that doesn't need to be consigned to the bitbucket of /dev/null just yet! ■

ASCII and EBCDIC

Although we refer to *ASCII* text (since that is what most Linux users will be using), UUE permits the text to be encoded in EBCDIC (a character set used by IBM mainframes). Fortunately, this is a much rarer case, and so doesn't need to be considered here.

THE AUTHOR

Steven Goodwin is a lead programmer, who has just finished off his fifth computer game. He has had more bugs than you've had hot dinners... When not working he can often be found relaxing at London LONIX meetings.