

**hexdump** — Display hexadecimal report on file

<a href="#">Description</a> <a href="#">Remarks and examples</a>	<a href="#">Syntax</a> <a href="#">Stored results</a>	<a href="#">Options</a> <a href="#">Also see</a>
---	--	---

## Description

`hexdump` displays a hexadecimal dump of a file or, optionally, a report analyzing the dump.

## Syntax

```
hexdump filename [ , options ]
```

<i>options</i>	Description
<u>a</u> nalyze	display a report on the dump rather than the dump itself
<u>t</u> abulate	display a full tabulation of the ASCII and extended ASCII characters in the <code>analyze</code> report
<u>n</u> oextended	do not display printable extended ASCII characters
<u>r</u> esults	store results containing the frequency with which each character code was observed; programmer's option
<u>f</u> rom(#)	dump or analyze first byte of the file; default is to start at first byte, <code>from(0)</code>
<u>t</u> o(#)	dump or analyze last byte of the file; default is to continue to the end of the file

## Options

`analyze` specifies that a report on the dump, rather than the dump itself, be presented.

`tabulate` specifies in the `analyze` report that a full tabulation of the ASCII and extended ASCII characters also be presented.

`noextended` specifies that `hexdump` not display printable extended ASCII characters, characters in the range 161–254 or, equivalently, 0xa1–0xfe. (`hexdump` does not display characters 128–160 and 255.)

`results` is for programmers. It specifies that, in addition to other stored results, `hexdump` store `r(c0)`, `r(c1)`, ..., `r(c255)`, containing the frequency with which each character code was observed.

`from(#)` specifies the first byte of the file to be dumped or analyzed. The default is to start at the first byte of the file, `from(0)`.

`to(#)` specifies the last byte of the file to be dumped or analyzed. The default is to continue to the end of the file.

## Remarks and examples

`hexdump` is useful when you are having difficulty reading a file with `infile`, `infix`, or `import delimited`. Sometimes, the reason for the difficulty is that the file does not contain what you think it contains, or that it does contain the format you have been told, and looking at the file in text mode is either not possible or not revealing enough.

Pretend that we have the file `myfile.raw` containing

```
Datsun 210      4589 35 5 1
VW Scirocco    6850 25 4 1
Merc. Bobcat   3829 22 4 0
Buick Regal    5189 20 3 0
VW Diesel      5397 41 5 1
Pont. Phoenix  4424 19 . 0
Merc. Zephyr   3291 20 3 0
Olds Starfire  4195 24 1 0
BMW 320i       9735 25 4 1
```

We will use `myfile.raw` with `hexdump` to produce output that looks like the following:

```
. hexdump myfile.raw
```

address	hex representation										character representation							
	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	0123456789abcdef	
0	44	61	74	73	75	6e	20	32	31	30	20	20	20	20	34		Datsun 210	4
10	35	38	39	20	20	33	35	20	20	35	20	20	31	0a	56	57	589 35 5 1.VW	
20	20	53	63	69	72	6f	63	63	6f	20	20	20	20	36	38	35	Scirocco 685	
30	30	20	20	32	35	20	20	34	20	20	31	0a	4d	65	72	63	0 25 4 1.Merc	
40	2e	20	42	6f	62	63	61	74	20	20	20	33	38	32	39	20	. Bobcat 3829	
50	20	32	32	20	20	34	20	20	30	0a	42	75	69	63	6b	20	22 4 0.Buick	
60	52	65	67	61	6c	20	20	20	20	35	31	38	39	20	20	32	Regal 5189 2	
70	30	20	20	33	20	20	30	0a	56	57	20	44	69	65	73	65	0 3 0.VW Diese	
80	6c	20	20	20	20	20	20	35	33	39	37	20	20	34	31	20	1 5397 41	
90	20	35	20	20	31	0a	50	6f	6e	74	2e	20	50	68	6f	65	5 1.Pont. Phoe	
a0	6e	69	78	20	20	34	34	32	34	20	20	31	39	20	20	2e	nix 4424 19 .	
b0	20	20	30	0a	4d	65	72	63	2e	20	5a	65	70	68	79	72	0.Merc. Zephyr	
c0	20	20	20	33	32	39	31	20	20	32	30	20	20	33	20	20	3291 20 3	
d0	30	0a	4f	6c	64	73	20	53	74	61	72	66	69	72	65	20	0.Olds Starfire	
e0	20	34	31	39	35	20	20	32	34	20	20	31	20	20	30	0a	4195 24 1 0.	
f0	42	4d	57	20	33	32	30	69	20	20	20	20	20	20	20	39	BMW 320i 9	
100	37	33	35	20	20	32	35	20	20	34	20	20	20	31	0a		735 25 4 1.	

hexdump can also produce output that looks like the following:

```
. hexdump myfile.raw, analyze
```

Line-end characters		Line length (tab=1)	
\r\n (Windows)	0	minimum	29
\r by itself (Mac)	0	maximum	29
\n by itself (Unix)	9		
Space/separator characters		Number of lines	9
[blank]	99	EOL at EOF?	yes
[tab]	0		
[comma] (,)	0	Length of first 5 lines	
Control characters		Line 1	29
binary 0	0	Line 2	29
CTL excl. \r, \n, \t	0	Line 3	29
DEL	0	Line 4	29
Extended (128-159,255)	0	Line 5	29
ASCII printable			
A-Z	20		
a-z	61	File format	ASCII
0-9	77		
Special (!@#\$ etc.)	4		
Extended (160-254)	0		
<b>Total</b>	<b>270</b>		

Observed were:

```
\n blank . 0 1 2 3 4 5 6 7 8 9 B D M 0 P R S V W Z a b c d e f g h i k l
n o p r s t u x y
```

Of the two forms of output, the second is often the more useful because it summarizes the file, and the length of the summary is not a function of the length of the file. Here is the summary for a file that is just over 4 MB long:

```
. hexdump bigfile.raw, analyze
```

Line-end characters		Line length (tab=1)	
\r\n (Windows)	147,456	minimum	29
\r by itself (Mac)	0	maximum	30
\n by itself (Unix)	2		
Space/separator characters		Number of lines	147,458
[blank]	1,622,039	EOL at EOF?	yes
[tab]	0		
[comma] (,)	0	Length of first 5 lines	
Control characters		Line 1	30
binary 0	0	Line 2	30
CTL excl. \r, \n, \t	0	Line 3	30
DEL	0	Line 4	30
Extended (128-159,255)	0	Line 5	30
ASCII printable			
A-Z	327,684		
a-z	999,436	File format	ASCII
0-9	1,261,587		
Special (!@#\$ etc.)	65,536		
Extended (160-254)	0		
<b>Total</b>	<b>4,571,196</b>		

Observed were:

```
\n \r blank . 0 1 2 3 4 5 6 7 8 9 B D M 0 P R S V W Z a b c d e f g h i
k l n o p r s t u x y
```

## 4 hexdump — Display hexadecimal report on file

Here is the same file but with a subtle problem:

```
. hexdump badfile.raw, analyze
Line-end characters                               Line length (tab=1)
  \r\n      (Windows)                          147,456      minimum                30
  \r by itself (Mac)                            0           maximum                90
  \n by itself (Unix)                           0
Space/separator characters                      Number of lines      147,456
 [blank]                                         1,622,016      EOL at EOF?          yes
 [tab]                                           0
 [comma] (,)                                    0           Length of first 5 lines
Control characters                               Line 1              30
 binary 0                                       8           Line 2              30
 CTL excl. \r, \n, \t                          4           Line 3              30
 DEL                                             0           Line 4              30
 Extended (128-159,255)                        24          Line 5              30
ASCII printable
 A-Z                                           327,683
 a-z                                           999,426      File format          BINARY
 0-9                                           1,261,568
 Special (!@#$ etc.)                          65,539
 Extended (160-254)                            16
-----
Total                                         4,571,196
Observed were:
 \0 ^C ^D ^G \n \r ^U blank & . 0 1 2 3 4 5 6 7 8 9 B D E M O P R S U V W
 Z a b c d e f g h i k l n o p r s t u v x y } ~ ^A E^C E^I E^M E^P
 ë é ö 255
```

In the above, the line length varies between 30 and 90 (we were told that each line would be 30 characters long). Also the file contains what `hexdump, analyze` labeled control characters. Finally, `hexdump, analyze` declared the file to be BINARY rather than ASCII.

We created the second file by removing two valid lines from `bigfile.raw` (60 characters) and substituting 60 characters of binary junk. We would defy you to find the problem without using `hexdump, analyze`. You would succeed, but only after much work. Remember, this file has 147,456 lines, and only two of them are bad. If you print 1,000 lines at random from the file, your chances of listing the bad part are only 0.013472. To have a 50% chance of finding the bad lines, you would have to list 52,000 lines, which is to say, review about 945 pages of output. On those 945 pages, each line would need to be drawn at random. More likely, you would list lines in groups, and that would greatly reduce your chances of encountering the bad lines.

The situation is not as dire as we make it out to be because, were you to read `badfile.raw` by using `infile`, it would complain, and here it would tell you exactly where it was complaining. Still, at that point you might wonder whether the problem was with how you were using `infile` or with the data. Moreover, our 60 bytes of binary junk experiment corresponds to transmission error. If the problem were instead that the person who constructed the file constructed two of the lines differently, `infile` might not complain, but later you would notice some odd values in your data (because obviously you would review the summary statistics, right?). Here `hexdump, analyze` might be the only way you could prove to yourself and others that the raw data need to be reconstructed.

## □ Technical note

In the full hexadecimal dump,

```
. hexdump myfile.raw
```

address	hex representation												character representation																			
	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	44	61	74	73	75	6e	20	32	31	30	20	20	20	20	34		D	a	t	s	u	n		2	1	0						4
10	35	38	39	20	20	33	35	20	20	35	20	20	31	0d	0a	56		5	8	9		5		1	.	.	V					
20	57	20	53	63	69	72	6f	63	63	6f	20	20	20	20	36	38		W		S	c	i	r	o	c	c	o				6	8
30	35	30	20	20	32	35	20	20	34	20	20	31	0d	0a	4d	65		5	0		2	5		4		1	.	.	M	e		

(output omitted)

addresses (listed on the left) are listed in hexadecimal. Above, 10 means decimal 16, 20 means decimal 32, and so on. Sixteen characters are listed across each line.

In some other dump, you might see something like

```
. hexdump myfile2.raw
```

address	hex representation												character representation																				
	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	
0	44	61	74	73	75	6e	20	32	31	30	20	20	20	20	34		D	a	t	s	u	n		2	1	0						4	
10	35	38	39	20	20	33	35	20	20	35	20	20	31	20	20		5	8	9		5		1										
20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20																		
*																																	
160	20	20	20	20	0a	56	57	20	53	63	69	72	6f	63				.	V		S	c	i	r	o	c							
170	63	6f	20	20	36	38	35	30	20	20	32	35	20	20			c	o		6	8	5	0		2	5							

(output omitted)

The \* in the address field indicates that the previous line is repeated until we get to hexadecimal address 160 (decimal 352).

□

## Stored results

`hexdump`, `analyze` and `hexdump`, results store the following in `r()`:

### Scalars

<code>r(Windows)</code>	number of <code>\r\n</code>
<code>r(Mac)</code>	number of <code>\r</code> by itself
<code>r(Unix)</code>	number of <code>\n</code> by itself
<code>r(blank)</code>	number of blanks
<code>r(tab)</code>	number of tab characters
<code>r(comma)</code>	number of comma (,) characters
<code>r(ct1)</code>	number of binary 0s; A–Z, excluding <code>\r</code> , <code>\n</code> , <code>\t</code> ; DELs; and 128–159, 255
<code>r(uc)</code>	number of A–Z
<code>r(lc)</code>	number of a–z
<code>r(digit)</code>	number of 0–9
<code>r(special)</code>	number of printable special characters (!@#, etc.)
<code>r(extended)</code>	number of printable extended characters (160–254)
<code>r(filesize)</code>	number of characters
<code>r(lmin)</code>	minimum line length
<code>r(lmax)</code>	maximum line length
<code>r(lnum)</code>	number of lines
<code>r(eoleof)</code>	1 if EOL at EOF, 0 otherwise
<code>r(l1)</code>	length of 1st line
<code>r(l2)</code>	length of 2nd line
<code>r(l3)</code>	length of 3rd line
<code>r(l4)</code>	length of 4th line
<code>r(l5)</code>	length of 5th line
<code>r(c0)</code>	number of binary 0s (results only)
<code>r(c1)</code>	number of binary 1s (^A) (results only)
<code>r(c2)</code>	number of binary 2s (^B) (results only)
...	...
<code>r(c255)</code>	number of binary 255s (results only)

### Macros

<code>r(format)</code>	ASCII, EXTENDED ASCII, or BINARY
------------------------	----------------------------------

## Also see

[D] [filefilter](#) — Convert ASCII or binary patterns in a file

[D] [type](#) — Display contents of a file