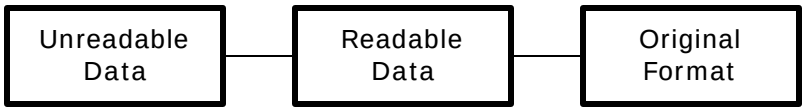
What is Base64?

Base64 is a different way of interpreting bits of data in order to transmit that data over a text-only medium, such as the body of an e-mail.

In the standard 8-bit ASCII character set, there are 256 characters that are used to format text. However, only a fraction of these characters are actually printable and readable when you are looking at them onscreen, or sending them in an e-mail.

We need a way to convert unreadable characters into readable characters, do something with them (i.e. send them in an e-mail), and convert them back to their original format.



So how do you convert unreadable, nonprintable characters into readable, printable characters? There are many ways to do this, but the way we are covering now is by using base64 encoding.

The 256 characters in the ASCII character set are numbered 0 through 255. For the tech savvy, this is the same as 2^8, 8 binary placeholders, or a byte. So for any ASCII character, you simply need one byte to represent this data. As far as a computer is concerned, there is no difference between an ASCII character, and a number between 0 and 255 (which is a string of 8 binary placeholders), only how it is interpreted. Because we are now detached from ASCII characters, you can also apply these same techniques to binary data, for example, a picture, or executable file. All you are doing is interpreting data one byte at a time.

The problem with representing data one byte at a time in a **readable** manner is that there are not 256 readable characters in the ASCII character set, so we cannot print a character for each of the 256 combinations that a byte can offer. So we need to take a different approach to looking at the bits in a byte.

So what if instead of looking at a whole byte, we looked at half of a byte, or 4 bits (also known as a nibble) at a time. This would be entirely possible because 2^4 is equal to 16, and there are certainly sixteen readable characters that we could use to represent each variation of nibble. This type of translation is known as hex. See Table 1 for the hex character set.

Binary	Decimal	Hex	Binary	Decimal	Hex
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	10	A
0011	3	3	1011	11	B
0100	4	4	1100	12	C
0101	5	5	1101	13	D
0110	6	6	1110	14	E
0111	7	7	1111	15	F

Table 1

The problem with using hex, is that since you are using one ASCII character (which is, remember, one byte long in storage space) to represent every four bits, anything you translate into hex will be exactly twice as big as the original data. This might not seem like a problem for a small message, but imagine you are trying to send an image or executable. The original size of perhaps a megabyte or more is now doubled. Sending this over email or a slow Internet connection will take twice as long.

Base64 As An Alternative

We now know that using 16 different characters to represent each half byte is a viable option, but not our ideal option because it is only half as space efficient as a byte. So how else can we dice bytes up to get our goal: readable characters for any value of 0 to 255?

Instead of looking at one byte at a time, and trying to chop that byte up, take several bytes and see what we can do with them.

Byte 1	Byte 2	Byte 3
0000 0000	0000 0000	0000 0000

Table 2

As you can easily see, using three bytes, we have a total of 24 bits. How else can we chop 24 bits up? If instead of 3 bytes of 8 bits each we use 4 "clumps" of 6 bytes each, what are we left with? Now we have 2^6 which equals 64. So now instead of needing 3 instances of a character that can represent any of 256 different combinations, we now need just 4 instances of a character that can represent any of 64 different combinations. The same bits as in the above table fit into the table below.

Clump 1	Clump 2	Clump 3	Clump 4
000000	000000	000000	000000

Table 3

Now we have to ask ourselves, "do we have 64 readable characters?". The answer is yes. The characters we will use are uppercase A-Z (26 characters), lowercase a-z (26 characters), 0-9 (10 characters), '+' (1 character) and '/' (1 character). 26 + 26 + 10 + 1 + 1 = 64, just the number we need. As you can surmise, base64 is still less space efficient than using a full byte, but instead of hex's double space usage, base64 uses only one and a third as much space. In other words for every 3 bytes, you must have 4 base64 characters. All of the characters listed above are easily readable.

The Mechanics of Base64: Encoding

How does the actual translation from bytes to base64 characters occur? We must first set up a mapping of values (0 through 63) to base64 characters (A-Z, a-z, 0-9, '+', and '/'). We can do this by creating a character array using the values from Table 4.

```
char base64Chars[] = {‘A’, ‘B’, ‘C’, ..., ‘9’, ‘+’, ‘/’};
```

Value:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
base64:	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
Value:	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
base64:	Q	R	S	T	U	V	W	X	Y	Z	a	b	c	d	e	f
Value:	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
base64:	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v
Value:	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
base64:	w	x	y	z	0	1	2	3	4	5	6	7	8	9	+	/

Table 4

For example, for a value of 25 (which is 011001 in binary) the base64 character would be 'Z', the character for binary 101010 (which is 42 in decimal) the base64 character would be 'q'.

```
char tempBase64Char;
tempBase64Char = base64Chars[25];    // tempBase64Char is now ‘Z’
tempBase64Char = base64Chars[42];    // tempBase64Char is now ‘q’
```

Let's start with something simple, a text-to-base64 conversion. We will convert the string "Hello World!" to a base64 representation. We will start by getting the ASCII and binary byte values for each letter. (See Table 5).

Character:	H	e	l	L	o	[space]
ASCII Value:	72	101	108	108	111	32
Binary Value:	01001000	01100101	01101100	01101100	01101111	00100000
Character:	W	o	R	L	d	!
ASCII Value:	87	111	114	108	100	33
Binary Value:	01010111	01101111	01110010	01101100	01100100	00100001

Table 5

Remember that for base64, we will be using three bytes at a time. Each ASCII character is one byte, so we will be working with "Hel", "lo[space]", "Wor", and "ld!" separately.

Let's start with the first three characters:

1. Convert the characters to binary.
2. "Hel" is 01001000 01100101 01101100 in binary. (Notice that there are 24 bits).
3. Convert the 24 bits from three 8 bit groups to four 6 bit groups. 01001000 01100101 01101100 becomes 010010 000110 010101 101100.
4. Convert each of the four 6 bit groups into decimal.
   010010 = 18
   000110 = 6
   010101 = 21
   101100 = 44
5. Use each of the four decimals to look up the base64 character code.
   18 = 'S'
   6 = 'G'
   21 = 'V'
   44 = 's'
6. You now have your first three ASCII characters ("Hel") encoded as base64 ("SGVs").

Follow these steps for the next 9 ASCII characters and you get the following results:
"Hel" = SGVs
"lo[space]" = bG8g
"Wor" = V29y
"ld!" = bGQh

The phrase "Hello World!" has been converted to "SGVsbG8gV29ybGQh". The original phrase has exactly 12 ASCII characters, and is represented by 16 base64 characters, exactly one and one third more than the original text.

So what happens, you might ask, if you don't have exact sets of three bytes? What if you had remainder bytes left over? For example, what if the data you had was "Hello" (5 ASCII characters, 5 bytes)? What if it was "blue" (4 ASCII characters, 4 bytes)? In those cases, you have groups of less than three letters: "Hello" groups into "Hel" "lo", and "blue" groups into "blu" "e".

To handle these cases, we throw one more readable character into our base64 character list. This character is not in the lookup table because it is only reserved for the two cases where you have one or two remainder bytes after grouping. We use the "=" character. Let's start with "Hello".

Character:	H	E	l	l	o
ASCII Value:	72	101	108	108	111
Binary Value:	01001000	01100101	01101100	01101100	01101111

Table 6

Follow the same exact steps for the first three characters as above. Your first three ASCII characters "Hel" are the same base64 as before "SGVs". For the remaining 2 characters, follow these steps:

1. Convert the characters into binary.
   "lo" is 01101100 01101111 in binary.
2. Starting from the left, separate the bytes into 6 bit chunks as best as possible.
   01101100 01101111 becomes 011011 000110 1111.

As you can see, we still need two more bits for the last group, plus a whole other six bits for the full four base64 characters. What we need is something looking like 011011 000110 1111xx xxxxxx. We can convert 011011 and 000110 to decimal just fine.

011011 = 27
000110 = 6
1111xx = what?
xxxxxx = what?

To resolve this problem, we fill the last two bits of 1111xx with 0's, so 111100 = 60. We now have:

011011 = 27
000110 = 6
111100 = 60
xxxxxx = what?

Our base64 characters so far are "bG8". Since we are missing one **single complete** base64 character, we add one of our special "=" characters to the back to signify that we are missing one byte. Our complete converted base64 string is now "bg8=". So the word "Hello" translates to "SGVSbg8=".

We do the same thing for the word "blue", which is missing 2 bytes.

Character:	b	L	u	e
ASCII Value:	98	108	117	101
Binary Value:	01100010	01101100	01110101	01100101

The first three characters should be easy by now to convert. "blu" is 01100010 01101100 01110101. Translate that to 6 bit groups and you get 011000 100110 110001 110101. These convert to "Ymx1" in base64. Now you have one remaining character, "e". We do the exact same thing as last time. "e" in binary is 01100101. When you split that into four 6 bit groups, you get the following:

011001 = 25
01xxxx = what?
xxxxxx = what?
xxxxxx = what?

Fill the second group with 0's to be able to look it up. 011001 010000 xxxxxx xxxxxx becomes "ZQ". Because you were missing **two complete bytes**, add two of our special character on the end. So the letter "e" in ASCII becomes "ZQ==". The word "blue" becomes "Ymx1ZQ==".

Note: I said before that base64 encoding is one and one third larger than the byte

representation. In the cases were you are missing a byte, it is actually slightly more than this. The actual range is from exactly one and one third to one and one third plus two characters.

The Mechanics of Base64: Decoding

We will now tackle translating from base64 characters back into normal bytes. We will use the same mapping of values (0 through 63) to base64 characters (A-Z, a-z, 0-9, '+', and '/').

The reverse process is relatively simple now that we know how to perform the forward operation. Let's start with the base64 string "YmFzZTY0IGlzIGZ1biEh". Right now, that makes no sense. We begin the same way, by looking up the value for each base64 character.

Character:	Y	m	F	z
base64 Value:	24	38	5	51
Binary Value:	011000	100110	000101	110011
Character:	Z	T	Y	0
base64 Value:	25	19	24	52
Binary Value:	011001	010011	011000	110100
Character:	I	G	I	z
base64 Value:	8	6	37	51
Binary Value:	001000	000110	100101	110011
Character:	I	G	Z	1
base64 Value:	8	6	25	53
Binary Value:	001000	000110	011000	110101
Character:	b	i	E	h
base64 Value:	27	34	4	33
Binary Value:	011011	100010	000100	100001

Table 7

It is very important to remember that when you are encoding, you use 8 bits for each character, and when you are decoding you use 6 bits for each character!

Once again, we start by chopping it into smaller pieces and work on each piece. When we are decoding a base64 string into normal bytes, we use 4 characters at a time instead of the 3 we used when encoding. So our base64 string is broken up from "YmFzZTY0IGlzIGZ1biEh" into "YmFz", "ZTY0", "IGlz", "IGZ1", and "biEh". Instead of using a number to look up a base64 character, we are now using a base64 character to look up a number. Reference Table 4 for these values.

Lets start with our first group, "YmFz".

1. Convert the base64 characters to binary. (Remember to use 6 bit binary!) "YmFz" is 011000 100110 000101 110011 in binary.

2. Convert the 24 bits from four 6 bit groups to three 8 bit groups.
   011000 100110 000101 110011 becomes 01100010 01100001 01110011.
3. Convert each of the three 8 bit groups into decimal.
   01100010 = 98
   01100001 = 97
   01110011 = 115
4. Use each of the three decimals to look up the ASCII character for that value.
   98 = 'b'
   97 = 'a'
   115 = 's'
   You now have your first four base64 characters ("YmFz") decoded as ASCII ("bas").

Follow these steps for the next 16 base64 characters and you get the following results: "ZTY0" = "e64"
"IGlz" = " is"
"IGZ1" = " fu"
"biEh" = "n!!"
The encoded base64 string "YmFzZTY0IGlzIGZ1biEh" has been decoded to "base64 is fun!!".

We know how to encode bytes when we don't have exact groups of three to work with. But how do you decode base64 that has our special symbol, "="? It is very similar, you just have to remember the rules that caused us to use the "=". One thing before we get started: base64 encoded text will **always** be in groups of 4 base64 characters; if the number of base64 characters is not divisible by 4 with no remainder, then you have corrupted data.

Let's try decoding a base64 string that contains the "=" symbol. Our string this time will be "Li4ub3IgbWF5YmUgbm90Lg==". The first thing we do is divide this up into groups of four characters. "Li4ub3IgbWF5YmUgbm90Lg==" becomes "Li4u", "b3Ig", "bWF5", "YmUg", bm90", and "Lg==". The first five quartets are decoded in the exact same manner. We just need to learn what to do for the last quartet, "Lg==".

Remember what the "="s mean: one "=" means that we were missing one whole byte when we encoded the data, two "="s means that we were missing two whole bytes when we encoded the data. We begin in the same way as before.

1. Begin by converting the base64 characters to their base64 values.
   'L' = 11
   'g' = 32
   '=' = nothing
   '=' = nothing
2. Convert the values to binary.
   11 = 001011
   32 = 100000
   nothing = xxxxxx (just to call it something)
   nothing = xxxxxx (just to call it something)
3. Convert the four 6 bit groups into three 8 bit groups.
   001011 100000 xxxxxx xxxxxx becomes 00101110 0000xxxx xxxxxxxx.

We know that because we had two "="s at the end, that we were missing two complete bytes in the original data. Remember where we had to add zeros when we encoded into base64? Those are the zeros you see in the second 8 bit group ("0000xxxx"). Because each of these 8 bit groups represents one byte from the original data, and we know that we are missing two whole bytes, we discard the last two 8 bit groups, "0000xxxx" and "xxxxxxx". So the only data we now need to worry about is the first byte, 00101110. We convert this value to decimal.

00101110 = 46

We convert the 46 to ASCII and we get the character '.' and add this to the other data that we have decoded.

"Li4u" = "..."
"b3Ig" = "or "
"bWF5" = "may"
"YmUg" = "be "
"bm90" = "not"
"Lg==" = "."

Our final decoded string is "...or maybe not."

Try It Yourself

(Use http://www.aardwulf.com/tutor/base64/base64.html to check your answers)
Here are a handful of examples for you to play around with. For problems 1-3, follow these steps:

1. Convert each character from ASCII into binary bytes (using 8 bits per character). Make sure to get the correct case of the letter. (Use Table 8.)
2. Divide your bytes into groups of three bytes.
3. Chop each group of three bytes into 4 six bit clumps.
4. Convert each clump into decimal.
5. Look up the corresponding base64 character from Table 4.

Problem 1:
Convert the following into base64:

aardwulf.com is great

Your Answer:

Problem 2:
Convert the following into base64:

Zeros and ones make no sense.

Your Answer:

Problem 3:
Convert the following into base64:

A man, a plan, a canal. Panama!

Your Answer:

For problems 4-6, follow these steps:

1. Convert each character from base64 to a six bit clump. Remember special rules about the '=' character. (Use Table 4.)
2. Divide your clumps into groups of four clumps of bits (24 bits each group).
3. Divide each group into 3 eight bit bytes.
4. Convert each byte into decimal.
5. Look up the corresponding ASCII character from Table 8.

Problem 4:

Convert the following into normal ASCII text:

WW91IGFyZSBkb2luZyB2ZXJ5IHdlbGwh

Your Answer:

Problem 5:

Convert the following into normal ASCII text:

SnVzdCBvbmUgbW9yZSBsZWZ0IHRvIGRlY29kZS4=

Your Answer:

Problem 6:

Convert the following into normal ASCII text:

Q29uZ3JhdHVsYXRpb25zISBZb3UgYXJlIG5vdyBhIGJhc2U2NCBndXJ1IQ==

Your Answer:

7 bit Original ASCII (readable characters)							
Decimal	**Character**	**Hex**	**Binary**	**Decimal**	**Character**	**Hex**	**Binary**
0-31	Unreadable Characters	00-1F	00000000-00011111				
32	(space)	20	00100000	80	P	50	01010000
33	!	21	00100001	81	Q	51	01010001
34	"	22	00100010	82	R	52	01010010
35	#	23	00100011	83	S	53	01010011
36	$	24	00100100	84	T	54	01010100
37	%	25	00100101	85	U	55	01010101
38	&	26	00100110	86	V	56	01010110
39	'	27	00100111	87	W	57	01010111
40	(	28	00101000	88	X	58	01011000
41	)	29	00101001	89	Y	59	01011001
42	*	2A	00101010	90	Z	5A	01011010
43	+	2B	00101011	91	[	5B	01011011
44	,	2C	00101100	92	\	5C	01011100
45	-	2D	00101101	93	]	5D	01011101
46	.	2E	00101110	94	^	5E	01011110
47	/	2F	00101111	95	_	5F	01011111
48	0	30	00110000	96	`	60	01100000
49	1	31	00110001	97	a	61	01100001
50	2	32	00110010	98	b	62	01100010
51	3	33	00110011	99	c	63	01100011
52	4	34	00110100	100	d	64	01100100
53	5	35	00110101	101	e	65	01100101
54	6	36	00110110	102	f	66	01100110
55	7	37	00110111	103	g	67	01100111
56	8	38	00111000	104	h	68	01101000
57	9	39	00111001	105	i	69	01101001
58	:	3A	00111010	106	j	6A	01101010
59	;	3B	00111011	107	k	6B	01101011
60	<	3C	00111100	108	l	6C	01101100
61	=	3D	00111101	109	m	6D	01101101
62	>	3E	00111110	110	n	6E	01101110
63	?	3F	00111111	111	o	6F	01101111
64	@	40	01000000	112	p	70	01110000
65	A	41	01000001	113	q	71	01110001
66	B	42	01000010	114	r	72	01110010
67	C	43	01000011	115	s	73	01110011
68	D	44	01000100	116	t	74	01110100
69	E	45	01000101	117	u	75	01110101
70	F	46	01000110	118	v	76	01110110
71	G	47	01000111	119	w	77	01110111
72	H	48	01001000	120	x	78	01111000
73	I	49	01001001	121	y	79	01111001
74	J	4A	01001010	122	z	7A	01111010
75	K	4B	01001011	123	{	7B	01111011
76	L	4C	01001100	124	|	7C	01111100
77	M	4D	01001101	125	}	7D	01111101
78	N	4E	01001110	126	~	7E	01111110
79	O	4F	01001111	127	Unreadable Character	7F	01111111

Table 8