



## Lic. Ciências da Computação

1º ano

2007/08

A.J.Proença

### Tema

## Introdução aos Sistemas de Computação



## Estrutura do tema ISC

1. Representação de informação num computador
2. Organização e estrutura interna dum computador
3. Execução de programas num computador
4. O processador e a memória num computador
5. Da comunicação de dados às redes

## Noção de computador (1)



### Um computador é um sistema que:

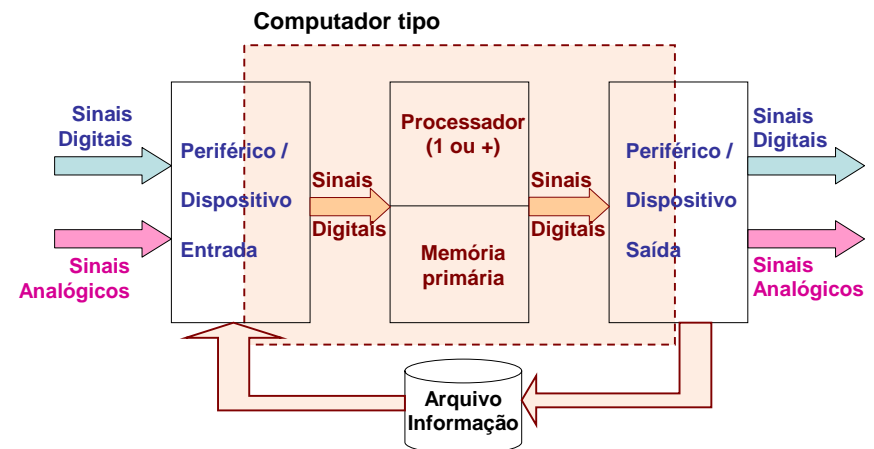
- recebe informação,  
processa / arquiva informação,  
transmite informação, e ...
- é programável  
i.e., a funcionalidade do sistema pode ser modificada,  
sem alterar fisicamente o sistema

Quando a funcionalidade é fixada no fabrico do sistema onde o computador se integra, diz-se que o computador existente nesse sistema está “embebido”: ex. telemóvel, máq. fotográfica digital, automóvel, ...

Como se representa a informação num computador ?

Como se processa a informação num computador ?

## Noção de computador (2)





- Como se representa a informação num computador ?
  - representação da informação num computador ->
- Como se processa a informação num computador ?
  - organização e funcionamento de um computador ->



## Como se representa a informação?

- com binary digits! (ver sistemas de numeração...)

## Tipos de informação a representar:

- textos (caracteres alfanuméricos)
  - » Baudot, Braille, ASCII, Unicode, ...
- números (para cálculo)
  - » inteiros: S+M, Compl. p/ 1, Compl. p/ 2, Excesso
  - » reais (fp): norma IEEE 754
- conteúdos multimédia
  - » imagens fixas: BMP, JPEG, GIF, PNG, ...
  - » audio-visuais: AVI, MPEG/MP3, ...
- código para execução no computador
  - » noção de *instruction set*

## Ex.: sistemas de numeração



**1532<sub>6</sub>** (base 6)

$$1 \cdot 6^3 + 5 \cdot 6^2 + 3 \cdot 6^1 + 2 \cdot 6^0 = 416_{10}$$

**1532.64<sub>10</sub>** (base 10)

$$1 \cdot 10^3 + 5 \cdot 10^2 + 3 \cdot 10^1 + 2 \cdot 10^0 + 6 \cdot 10^{-1} + 4 \cdot 10^{-2} = 1532.64_{10}$$

**1532<sub>13</sub>** (base 13)

$$1 \cdot 13^3 + 5 \cdot 13^2 + 3 \cdot 13^1 + 2 \cdot 13^0 = 3083_{10}$$

**110110.011<sub>2</sub>** (base 2)

$$1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} = 54.375_{10}$$

## Ex.: representação de texto com ASCII (7 bits)

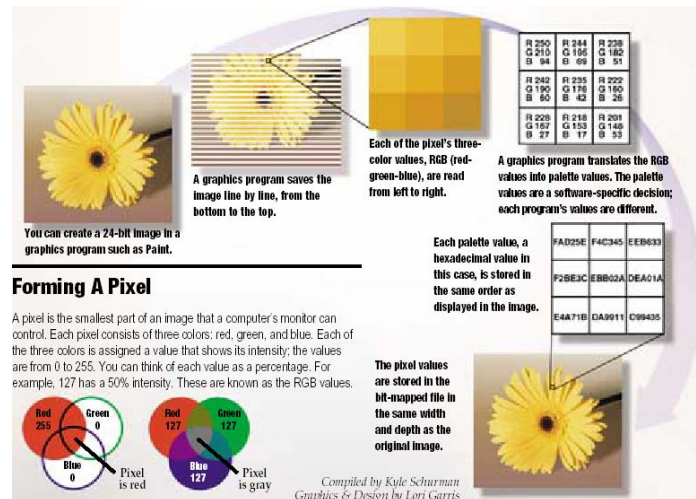


### Tabela ASCII 7 bits

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

H	e	l	l	o		w	o	r	l	d	!
48	65	6c	6c	6f	20	77	6f	72	6c	64	21

## Ex.: representação de uma imagem em bitmap



AJProença, Sistemas de Computação, UMinho, 2007/08

9

## Ex.: representação de código para execução num PC

```
int x = x+y;
```

- Código numa linguagem de programação
  - somar 2 inteiros

```
addl 8(%ebp),%eax
```

- Código numa linguagem mais próxima do processador

Idêntico à expressão

$x = x + y$

- somar 2 inteiros (de 4-bytes)
- operandos:
  - x: no registo `eax`
  - y: na memória em `[(ebp)+8]`

```
0x401046: 03 45 08
```

- Código “objecto” (em hexadecimal)
  - instrução com 3-bytes
  - na memória em `0x401046`

AJProença, Sistemas de Computação, UMinho, 2007/08

10

## Representação da informação num computador (2)

### Tipos de informação a representar:

- textos (caracteres alfanuméricos)
  - » [Baudot](#), [Braille](#), [ASCII](#), [Unicode](#) ([charts](#)), ...
- números (para cálculo)
  - » inteiros: S+M, Compl. p/ 1, Compl. p/ 2, Excesso
  - » reais (*f*p): norma IEEE 754
- conteúdos multimédia
  - » imagens fixas: [BMP](#), [JPEG](#), [GIF](#), [PNG](#), ...
  - » audio-visuais: [AVI](#), [MPEG/MP3](#), ...
- código para execução no computador
  - » noção de *instruction set*

AJProença, Sistemas de Computação, UMinho, 2007/08

11

## Caracterização dos ficheiros com documentos electrónicos (1)

### Elementos num documento electrónico:

- texto codificado (ASCII, Unicode, ...)
- especificação de formatação (margens, estilos, ...)
- tabelas e gráficos (directas, importadas, ligadas, ...)
- audiovisuais
  - desenhos e imagens
  - sons
  - vídeos
  - ...

AJProença, Sistemas de Computação, UMinho, 2007/08

12



## Tipos de ficheiros de acordo com o conteúdo:

### – apenas texto

- tipo de ficheiro: **\*.txt**
- formato do ficheiro: puro texto codificado em ASCII, Unicode, ...
- aplicação para o manusear/editar: editor de texto (NotePad, ...)

### – texto, mas com especificações para formatação

- tipos de ficheiro:
  - Rich Text Format (\*.rtf), proprietário (Microsoft)
  - Hyper-Text Markup Language (\*.html), standard
- formato do ficheiro: puro texto codificado em ASCII
- aplicações para o manusear/editar: processador de texto (Word, ...), editor de páginas Web (FrontPage, ...)

### – texto e imagens, apenas imagens ...



## Tipos de ficheiros de acordo com o conteúdo (cont.):

### – texto e imagens com codificação binária **proprietária**

- exemplos de tipos de ficheiro:
  - documentos Microsoft Word/Excel (\*.doc / \*.xls)
  - documentos/slides Microsoft PowerPoint (\*.ppt / \*.pps)
  - documentos Acrobat (\*.pdf)

### – apenas imagens com codificação específica

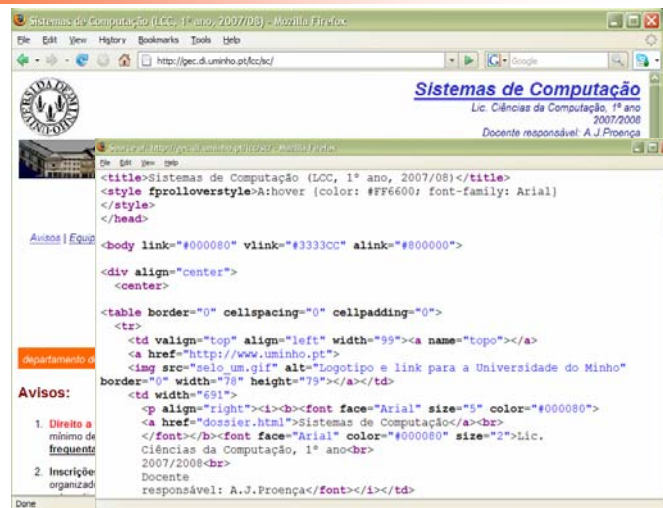
- exemplos de tipos de ficheiro / aplicações:
  - desenhos esquemáticos: qualquer aplicação de Office
  - gráficos a partir de tabelas: em folhas de cálculo (Excel, ...)
  - desenhos em formato vectorial (FreeHand, AutoCad, ...)
  - desenhos orientados ao pixel (CorelDraw, Photoshop, ...)

## Exemplos de documentos...



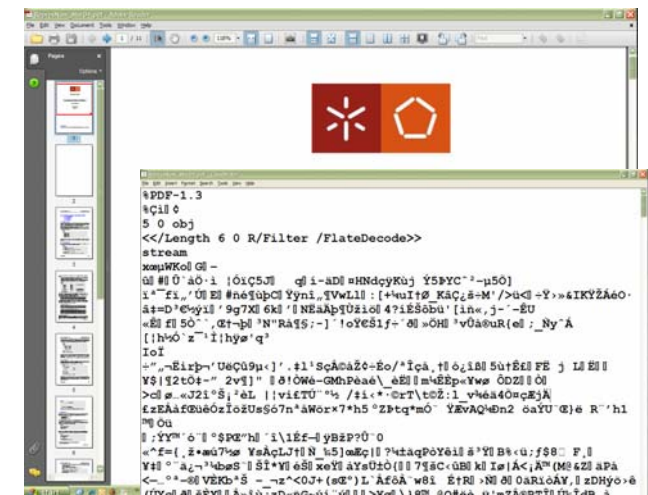
## Página da disciplina em HTML:

- interpretada pelo browser
- visualizada por um editor de texto



## Ficheiro com Notas de Estudo em PDF:

- visualizada com Acrobat Reader
- visualizada por um editor de texto



Search

Home

Contents

Feedback

Random

Baudot ==&gt;

# Baudot code

<[communications](#)> (For etymology, see [baud](#)) A [character set](#) predating [EBCDIC](#) and used originally and primarily on [paper tape](#). Use of Baudot reportedly survives in [TDDs](#) and some HAM radio applications.

In Baudot, characters are expressed using five [bits](#). Baudot uses two code sub-sets, the "letter set" (LTRS), and the "figure set" (FIGS). The FIGS character (11011) signals that the following code is to be interpreted as being in the FIGS set, until this is reset by the LTRS (11111) character.

binary	hex	LTRS	FIGS
-----			
00011	03	A	-
11001	19	B	?
01110	0E	C	:
01001	09	D	\$
00001	01	E	3
01101	0D	F	!
11010	1A	G	&
10100	14	H	#
00110	06	I	8
01011	0B	J	BELL
01111	0F	K	(
10010	12	L	)
11100	1C	M	.
01100	0C	N	,
11000	18	O	9
10110	16	P	0
10111	17	Q	1
01010	0A	R	4
00101	05	S	'
10000	10	T	5
00111	07	U	7
11110	1E	V	;
10011	13	W	2
11101	1D	X	/
10101	15	Y	6
10001	11	Z	"
01000	08	CR	CR
00010	02	LF	LF
00100	04	SP	SP
11111	1F	LTRS	LTRS
11011	1B	FIGS	FIGS
00000	00	[..unused..]	

Where CR is [carriage return](#), LF is [linefeed](#), BELL is the [bell](#), SP is space, and STOP is the stop character.

Note: these bit values are often shown in inverse order, depending (presumably) which side of the [paper tape](#) you were looking at.

Local implementations of Baudot may differ in the use of #, STOP, BELL, and '.

(1997-01-30)

Try this search on [OneLook](#) / [Google](#)

---

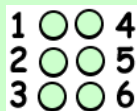
**Nearby terms:** [baud barf](#) « [Baudot](#) « [Baudotbetical order](#) « [Baudot code](#) » [baud rate](#) » [bawk](#) » [bay](#)

---

## ...Braille: Deciphering the Code...

Every character in the braille code is based on an arrangement of one to six raised dots. Each dot has a numbered position in the braille cell. These characters make up the letters of the alphabet, punctuation marks, numbers, and everything else you can do in print.

### The Braille Cell



The letter "A" is written with only 1 dot.



The letter "D" has dots 1, 4, and 5.



The letter "Y" has dots 1, 3, 4, 5, and 6.



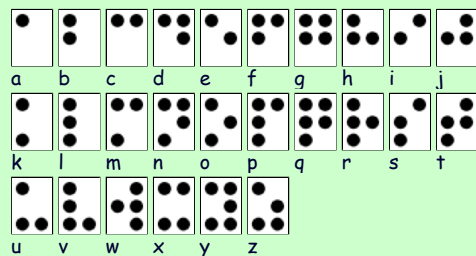
A "Period" is written with dots 2, 5, and 6. (Do you see how it is the same shape as the letter "D," only lower down in the cell?)



When all six dots are used, the character is called a "full cell"



The picture below shows you how the dots are arranged in the braille cell for each letter of the alphabet. See if you can find the letters in your name and tell the dot numbers for each one.



Braille does not have a separate alphabet of capital letters as there is in print. Capital letters are indicated by placing a dot 6 in front of the letter to be capitalized. Two capital signs mean the whole word is capitalized.

One Letter Capitalized

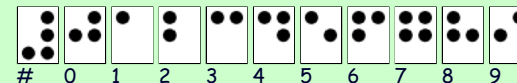


Entire Word Capitalized



### Braille Numbers

Braille numbers are made using the first ten letters of the alphabet, "a" through "j", and a special number sign, dots 3, 4, 5, and 6.



Larger numbers only need one number sign. The comma in braille is dot 2.



### Expanding the Code

Now that you understand how dots are arranged in the braille cell to make the letters of the alphabet and numbers, you're ready to learn more about the code. Braille uses special characters called contractions to make words shorter. We use contractions like "don't" as a short way of writing two words, such as "do" and "not." In braille there are many additional contractions, 189 in all! Using these contractions saves space, which is very important because braille books are much larger and longer than print books.

Some contractions stand for a whole word



Other contractions stand for a group of letters within a word.



In addition to contractions, the braille code includes short-form words which are abbreviated spellings of common longer words. For example, "tomorrow" is spelled "tm", "friend" is spelled "fr", and "little" is spelled "ll" in braille.

You might think that because short-form words are so easy to spell that children who write braille get a break on their spelling tests. Actually, braille readers also learn regular spelling for typing on a computer.

Let's see what kind of difference contractions make in braille. Look at the same phrase, **you like him**, in uncontracted braille (sometimes called "grade 1 braille") and contracted braille (sometimes called "grade 2 braille"). What do you notice about the length of the two phrases?

Uncontracted Braille:



Contracted Braille:



### Other Braille Codes

The braille code used for writing regular text in books, magazines, school reports, and letters is known as "literary braille." There are other codes, though, that let people who are blind write just about anything, from math problems to music notes to computer notation!





**JimPrice.Com**

**ALL NEW - [Connector Reference](#) and [Games](#).**

## ASCII Chart

Dogs come when they are called.  
Cats take a message and get back to you.  
-- Mary Bly

**ASCII** - The **A**merican **S**tandard **C**ode for Information Interchange is a standard seven-bit code that was proposed by [ANSI](#) in 1963, and finalized in 1968. Other sources also credit much of the work on ASCII to work done in 1965 by Robert W. Bemer ([www.bobbemer.com](http://www.bobbemer.com)). ASCII was established to achieve compatibility between various types of data processing equipment. Later-day standards that document ASCII include ISO-14962-1997 and ANSI-X3.4-1986(R1997).

ASCII, pronounced "ask-key", is the common code for microcomputer equipment. The standard ASCII character set consists of 128 decimal numbers ranging from zero through 127 assigned to letters, numbers, punctuation marks, and the most common special characters. The Extended ASCII Character Set also consists of 128 decimal numbers and ranges from 128 through 255 representing additional special, mathematical, graphic, and foreign characters.

Every now and again, I've wished that I had an ASCII chart handy, so I made one and stuck it on this page so that I could find it in a hurry. One thing led to another, and folks started asking me questions about ASCII and other character representations, so I've tried to update this page a bit to answer some of the most [common questions](#). Also, I've added additional info, such as [IBM PC Keyboard Scan Codes](#), and a list of [other references](#).

## My ASCII Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

Here's a link to a [decimal-to-ASCII](#) chart.

You may also want to read the [Q & A](#), below.

## ASCII Q & A

Every now and then, I get questions about ASCII. Here are a few:

**Q:** What does ASCII stand for?

**A:** ASCII stands for American Standard Code for Information Interchange.

**Q:** What do CR, LF, SO, and so forth mean?

**A:** A more detailed description of the first 32 characters can be found [here](#).

**Q:** Someone wants me to send my resume (or some other file) to them in ASCII, so what do I do?

**A:** In general, if somebody asks for your resume or another document "in ASCII", what they really want is a copy of your resume (or whatever the document is) in electronic form, but without any formatting. Save your resume (or other document) from the word processor you're using (such as Microsoft Word ® ) as plain text, and then paste it into an e-mail (or copy it to a floppy disk) and send it to whoever wanted it.

**Q:** OK, fine, but how do I save a Microsoft Word ® document as plain text?

**A:** Here are some simple instructions for saving a document as plain text in Microsoft Word® '97. The procedure is similar for other word processors.

1. Click on "File".
2. Select "Save As".
3. In the box marked "Save as type:", select "Text Only (\*.txt)" or "Text Only with Line Breaks (\*.txt)".
4. In the box marked "File name:", enter the file name (a different name than the one that you had been using).
5. Click on the button marked "Save".

If you get a warning indicating that formatting will be lost, don't be alarmed. The whole point of the process is to remove the formatting from the document before you send it.

**Q:** I'm writing a program that directly interfaces with the keyboard of an IBM PC, and I need to accept keystrokes for F1, F2, and other keys. What are the IBM PC keyboard codes for F1, F2, and ALT-A, etc?

**A:** See [IBM PC Keyboard Scan Codes](#), below

**A:** Also, if you're trying to enter CTRL-ALT-DEL, you might want to try one of my [reboot utilities](#).

**Q:** What are the ASCII codes for things like the degrees symbol (°), the trademark symbol, solid blocks and other special symbols?

**A:** The answer is "it depends". There's really not an ASCII character for most of the special characters (other than those shown in the [table](#) at the top of this page. However, there are different extended ASCII sets that include a lot more special characters. (See [IBM PC Extended ASCII](#), below.) Also, for information about including special symbols on a web page, you might look at some of the [other charts](#) and [references](#), below.

**Q:** How do I generate extended ASCII characters from the keyboard?

**A:** DOS will allow you to enter extended ASCII characters into many programs. Here's how you do it:

1. Make sure NumLock is enabled on your keyboard.
2. Press and hold the ALT key.
3. While holding down the ALT key, enter the 3-digit decimal code for the extended ASCII character you want to generate.
4. Release the ALT key.

The trick to this, of course, is knowing the decimal equivalent of the ASCII characters that you want to generate. Since the [extended ASCII Chart](#) is in hex, you probably want a [guide for to converting hex to decimal](#).

**Q:** Does this work in Windows?

**A:** A better way to enter special symbols in Windows applications (such as Word), is to take advantage of the symbols in the fonts provided by Windows. For example, in Word, use the following steps:

1. Click on "Insert" (in the menu at the top of the page)
2. From the drop-down menu, select "Symbol". A chart of special symbols will appear
3. From the chart, you can select a symbol to insert into your document by highlighting the symbol, and clicking on the word "Insert" at the bottom of the chart.

**Q:** What's the ASCII code for CTRL-ALT-DEL, and how do I put it in a DOS batch file?

**A:** There's not an easy way to get CTRL-ALT-DEL into a batch file, per se. However, I've created

several PC reboot utilities (for both DOS and Windows) that you can try. You can find them on my [shareware page](#).

**Q:** What's the ASCII code for CTRL-A, or CTRL-Z?

**A:** The value for CTRL-A (^A) is 01. The value for CTRL-Z (^Z), which is often used as an end-of-file marker in DOS is 26 (decimal). All the other CTRL-B through CTRL-Y characters fall in between. (CTRL-B is 2, CTRL-C is 3, and so forth.)

Control Codes

The following is a more detailed description of the first 32 ASCII characters, often referred to as control codes.

- NUL (null)
- SOH (start of heading)
- STX (start of text)
- ETX (end of text)
- EOT (end of transmission) - Not the same as ETB
- ENQ (enquiry)
- ACK (acknowledge)
- BEL (bell) - Caused teletype machines to ring a bell. Causes a beep in many common terminals and terminal emulation programs.
- BS (backspace) - Moves the cursor (or print head) move backwards (left) one space.
- TAB (horizontal tab) - Moves the cursor (or print head) right to the next tab stop. The spacing of tab stops is dependent on the output device, but is often either 8 or 10.
- LF (NL line feed, new line) - Moves the cursor (or print head) to a new line. On Unix systems, moves to a new line AND all the way to the left.
- VT (vertical tab)
- FF (form feed) - Advances paper to the top of the next page (if the output device is a printer).
- CR (carriage return) - Moves the cursor all the way to the left, but does not advance to the next line.
- SO (shift out) - Switches output device to alternate character set.
- SI (shift in) - Switches output device back to default character set.
- DLE (data link escape)
- DC1 (device control 1)
- DC2 (device control 2)
- DC3 (device control 3)
- DC4 (device control 4)
- NAK (negative acknowledge)
- SYN (synchronous idle)
- ETB (end of transmission block) - Not the same as EOT
- CAN (cancel)
- EM (end of medium)
- SUB (substitute)
- ESC (escape)
- FS (file separator)
- GS (group separator)
- RS (record separator)
- US (unit separator)

IBM PC Keyboard Scan Codes

For many of the special key combinations such as ALT-A, F1, PgUp, and so forth, the IBM PC uses a special two-character escape sequence. Depending on the programming language being used and the level at which the keyboard is being accessed, the escape character is either ESC (27, 0x1B), or NUL (0). Here are some common sequences:

Char.	Decimal Pair	Hex Pair	Char.	Decimal Pair	Hex Pair
ALT-A	(00,30)	(0x00,0x1e)	ALT-B	(00,48)	(0x00,0x30)
ALT-C	(00,46)	(0x00,0x2e)	ALT-D	(00,32)	(0x00,0x20)
ALT-E	(00,18)	(0x00,0x12)	ALT-F	(00,33)	(0x00,0x21)
ALT-G	(00,34)	(0x00,0x22)	ALT-H	(00,35)	(0x00,0x23)

ALT-I	(00,23)	(0x00,0x17)	ALT-J	(00,36)	(0x00,0x24)
ALT-K	(00,37)	(0x00,0x25)	ALT-L	(00,38)	(0x00,0x26)
ALT-M	(00,50)	(0x00,0x32)	ALT-N	(00,49)	(0x00,0x31)
ALT-O	(00,24)	(0x00,0x18)	ALT-P	(00,25)	(0x00,0x19)
ALT-Q	(00,16)	(0x00,0x10)	ALT-R	(00,19)	(0x00,0x13)
ALT-S	(00,31)	(0x00,0x1a)	ALT-T	(00,20)	(0x00,0x14)
ALT-U	(00,22)	(0x00,0x16)	ALT-V	(00,47)	(0x00,0x2f)
ALT-W	(00,17)	(0x00,0x11)	ALT-X	(00,45)	(0x00,0x2d)
ALT-Y	(00,21)	(0x00,0x15)	ALT-Z	(00,44)	(0x00,0x2c)
PgUp	(00,73)	(0x00,0x49)	PgDn	(00,81)	(0x00,0x51)
Home	(00,71)	(0x00,0x47)	End	(00,79)	(0x00,0x4f)
UpArrw	(00,72)	(0x00,0x48)	DnArrw	(00,80)	(0x00,0x50)
LftArrw	(00,75)	(0x00,0x4b)	RtArrw	(00,77)	(0x00,0x4d)
F1	(00,59)	(0x00,0x3b)	F2	(00,60)	(0x00,0x3c)
F3	(00,61)	(0x00,0x3d)	F4	(00,62)	(0x00,0x3e)
F5	(00,63)	(0x00,0x3f)	F6	(00,64)	(0x00,0x40)
F7	(00,65)	(0x00,0x41)	F8	(00,66)	(0x00,0x42)
F9	(00,67)	(0x00,0x43)	F10	(00,68)	(0x00,0x44)
F11	(00,133)	(0x00,0x85)	F12	(00,134)	(0x00,0x86)
ALT-F1	(00,104)	(0x00,0x68)	ALT-F2	(00,105)	(0x00,0x69)
ALT-F3	(00,106)	(0x00,0x6a)	ALT-F4	(00,107)	(0x00,0x6b)
ALT-F5	(00,108)	(0x00,0x6c)	ALT-F6	(00,109)	(0x00,0x6d)
ALT-F7	(00,110)	(0x00,0x6e)	ALT-F8	(00,111)	(0x00,0x6f)
ALT-F9	(00,112)	(0x00,0x70)	ALT-F10	(00,113)	(0x00,0x71)
ALT-F11	(00,139)	(0x00,0x8b)	ALT-F12	(00,140)	(0x00,0x8c)

Hint - If you look at how the keys are laid out on the keyboard, you'll probably see the pattern.

IBM PC Extended ASCII Display Characters

Strictly speaking, the ASCII character set only includes values up to 127 decimal (7F hex). However, when the IBM PC was developed, the video card contained one byte for each character in the 80x25 character display. Gee...what to do with that extra bit per character? Why not invent 128 new characters, for line-drawing and special symbols? The result, of course, was the extended ASCII character set for the IBM PC. The chart below shows (most of) the characters that can be generated by the display in the original IBM PC.

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
1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
2	3	4	5	6	7	8	9	A	B	C	D	E	F		
3	4	5	6	7	8	9	A	B	C	D	E	F			
4	5	6	7	8	9	A	B	C	D	E	F				
5	6	7	8	9	A	B	C	D	E	F					
6	7	8	9	A	B	C	D	E	F						
7	8	9	A	B	C	D	E	F							
8	9	A	B	C	D	E	F								
9	A	B	C	D	E	F									
A	B	C	D	E	F										
B	C	D	E	F											
C	D	E	F												
D	E	F													
E	F														
F															

Microsoft Windows ® has a different notion about what the high-order (upper 128) characters are, as shown in the table below.



	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																

Here's a link to a [decimal-to-Extended ASCII](#) chart.

## Converting Hex to Decimal

Here's a chart that shows the conversion between hex and decimal.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	000	001	002	003	004	005	006	007	008	009	010	011	012	013	014	015
1	016	017	018	019	020	021	022	023	024	025	026	027	028	029	030	031
2	032	033	034	035	036	037	038	039	040	041	042	043	044	045	046	047
3	048	049	050	051	052	053	054	055	056	057	058	059	060	061	062	063
4	064	065	066	067	068	069	070	071	072	073	074	075	076	077	078	079
5	080	081	082	083	084	085	086	087	088	089	090	091	092	093	094	095
6	096	097	098	099	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

If you're having trouble getting the hang of the above chart, here's a hint.  
Hex 41 (written as 0x41 in the programming language C) is equivalent to decimal 65.

## Converting Hex to Octal

Here's a chart that shows the conversion between hex and octal.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	000	001	002	003	004	005	006	007	010	011	012	013	014	015	016	017
1	020	021	022	023	024	025	026	027	030	031	032	033	034	035	036	037
2	040	041	042	043	044	045	046	047	050	051	052	053	054	055	056	057
3	060	061	062	063	064	065	066	067	070	071	072	073	074	075	076	077
4	100	101	102	103	104	105	106	107	110	111	112	113	114	115	116	117
5	120	121	122	123	124	125	126	127	130	131	132	133	134	135	136	137
6	140	141	142	143	144	145	146	147	150	151	152	153	154	155	156	157
7	160	161	162	163	164	165	166	167	170	171	172	173	174	175	176	177
8	200	201	202	203	204	205	206	207	210	211	212	213	214	215	216	217
9	220	221	222	223	224	225	226	227	230	231	232	233	234	235	236	237
A	240	241	242	243	244	245	246	247	250	251	252	253	254	255	256	257
B	260	261	262	263	264	265	266	267	270	271	272	273	274	275	276	277
C	300	301	302	303	304	305	306	307	310	311	312	313	314	315	316	317
D	320	321	322	323	324	325	326	327	330	331	332	333	334	335	336	337
E	340	341	342	343	344	345	346	347	350	351	352	353	354	355	356	357
F	360	361	362	363	364	365	366	367	370	371	372	373	374	375	376	377

If you're having trouble getting the hang of the above chart, here's a hint.  
Hex 41 (written as 0x41 in the programming language C) is equivalent to octal 101.

## Other ASCII Charts

Here are some other folks' ASCII charts, and other interesting ASCII-related references.

- [ISO 646](#)
- [ASCII Table.Com](#) - The Q and A looks familiar, doesn't it?
- [ASCII - ISO 8859-1 with HTML 3.0 Entities Table](#)
- [ASCII Character Set Revised, ANSI X3.110-1983](#)
- [ASCII Character Set, ANSI X3.4-1968](#)
- [ASCII Symbol Table - with HTML codes](#)
- [Decimal ASCII for HTML](#)
- [EBCDIC Character Set](#) - Not ASCII at all
- [BAUDOT](#) - Another Set of Character Codes
- [More on BAUDOT](#)
- [Brief History of Character Codes](#) - Good reference material!
- [Braille Chart](#) **NEW!** - A really neat resource.
- [Mac OS Character Representations](#) **NEW!** - For all you Mac programmers.


## Other Things You May Want to Know

- [What's its format?](#) **NEW!** - Almost every file format around.
- [Official US Time](#) **NEW!** - Current time, timezones, history, etc.
- [Connector Reference](#) **NEW!** - From NullModem.Com - DB-25, DB-9, Null Modem, more!
- [Common Terms](#) **NEW!** - From NullModem.Com - Ever-growing glossary of computer and electronics terms
- [Pro Sound References](#) **NEW!** - Info about pro sound, vintage consoles, cart machines and more!
- [Protocols.Com](#) - A good reference of protocols.
- [Area Code Finder](#)
- [Area Code Decoder](#)
- [555-1212](#) - Phone numbers and addresses for everyone!
- [KnowX.Com](#) - Great people-finder!
- [Currency Rates](#) - From Rubicon International
- [International Couriers](#) **NEW!**
- [Periodic Table of the Elements](#) **NEW!**
- [Basic Dictionary of ASL Terms](#) **NEW!** - American Sign Language info
- [Modem Command Sets](#) **NEW!** - Good reference on AT command sets.
- [Well-known IP Port Numbers](#)
- [Barcode Info](#)
- [BarcodesIsland.Com](#) - Lots of barcode info
- [Barcode Primer](#)
- [Barcode Formats](#)
- [Code 3 of 9](#)
- [More Barcode Formats](#)



This page and its contents, copyright © 1996-2002, [JimPrice.Com](#). Here's a link to our [privacy statement](#). If you have any questions about this page, please e-mail me at: [jim@jimprice.com](mailto:jim@jimprice.com).

This site last updated 8/12/2002.

**The Standard**

Home | Site Map | Search

**Contents**

[What Characters Does the Unicode Standard Include?](#)  
[Encoding Forms](#)  
[Defining Elements of Text](#)  
[Text Processing](#)  
[Interpreting Characters and Rendering Glyphs](#)  
[Character Sequences](#)  
[Principles of the Unicode Standard](#)  
[Assigning Character Codes](#)  
[Conformance to the Unicode Standard](#)  
[Stability](#)  
[Unicode and ISO/IEC 10646](#)  
[For Further Information](#)

**Related Links**

[Latest Version of the Unicode Standard](#)  
[The Unicode Standard, Version 3.0](#)  
[Chapter 1: Introduction](#)  
[Book Order Form](#)  
[FAQ](#)  
[Glossary](#)  
[Unicode Policies](#)  
[New in Version 3.0](#)  
[Updates and Errata](#)  
[Unicode-Enabled Products](#)  
[Contacting Unicode](#)

## The Unicode® Standard: A Technical Introduction

The Unicode Standard is the universal character encoding standard used for representation of text for computer processing. Versions of the Unicode Standard are fully compatible and synchronized with the corresponding versions of International Standard ISO/IEC 10646. For example, Unicode 3.0 contains all the same characters and encoding points as ISO/IEC 10646-1:2000. Unicode 3.1 adds all the characters and encoding points of ISO/IEC 10646-2:2001. The Unicode Standard provides additional information about the characters and their use. Any implementation that is conformant to Unicode is also conformant to ISO/IEC 10646.

**Note:**  
This is intended as a concise source of information about the Unicode® Standard. It is neither a comprehensive definition of, nor a technical guide to the Unicode Standard. Authoritative information can be found at [Latest Version of the Unicode Standard](#). That link will guide you both to the most recent major version, published as a book, and to the subsequent minor versions, published on the web.

Unicode provides a consistent way of encoding multilingual plain text and brings order to a chaotic state of affairs that has made it difficult to exchange text files internationally. Computer users who deal with multilingual text – business people, linguists, researchers, scientists, and others – will find that the Unicode Standard greatly simplifies their work. Mathematicians and technicians, who regularly use mathematical symbols and other technical characters, will also find the Unicode Standard valuable.

The design of Unicode is based on the simplicity and consistency of ASCII, but goes far beyond ASCII's limited ability to encode only the Latin alphabet. The Unicode Standard provides the capacity to encode all of the characters used for the written languages of the world. To keep character coding simple and efficient, the Unicode Standard assigns each character a unique numeric value and name.

The original goal was to use a single 16-bit encoding that provides code points for more than 65,000 characters. While 65,000 characters are sufficient for encoding most of the many thousands of characters used in major languages of the world, the Unicode standard and ISO/IEC 10646 now support three encoding forms that use a common repertoire of characters but allow for encoding as many as a million more characters. This is sufficient for all known character encoding requirements, including full coverage of all historic scripts of the world, as well as common notational systems.

### What Characters Does the Unicode Standard Include?

The Unicode Standard defines codes for characters used in all the major languages written today. Scripts include the European alphabetic scripts, Middle Eastern right-to-left scripts, and many scripts of Asia.

The Unicode Standard further includes punctuation marks, diacritics, mathematical symbols, technical symbols, arrows, dingbats, etc. It provides codes for diacritics, which are modifying character marks such as the tilde (~), that are used in conjunction with base characters to represent accented letters (ñ, for example). In all, the Unicode Standard, Version 3.2 provides codes for 95,221 characters from the world's alphabets, ideograph sets, and symbol collections.

The majority of common-use characters fit into the first 64K code points, an area of the codespace that is called the basic multilingual plane, or *BMP* for short. There are about 6,700 unused code points for future

expansion in the BMP, plus over 870,000 unused supplementary code points on the other planes. More characters are under consideration for addition to future versions of the standard.

The Unicode Standard also reserves code points for private use. Vendors or end users can assign these internally for their own characters and symbols, or use them with specialized fonts. There are 6,400 private use code points on the BMP and another 131,068 supplementary private use code points, should 6,400 be insufficient for particular applications.

## Encoding Forms

Character encoding standards define not only the identity of each character and its numeric value, or code point, but also how this value is represented in bits.

The Unicode Standard defines three encoding forms that allow the same data to be transmitted in a byte, word or double word oriented format (i.e. in 8, 16 or 32-bits per code unit). All three encoding forms encode the *same* common character repertoire and can be efficiently transformed into one another without loss of data. The Unicode Consortium fully endorses the use of any of these encoding forms as a conformant way of implementing the Unicode Standard.

UTF-8 is popular for HTML and similar protocols. UTF-8 is a way of transforming all Unicode characters into a variable length encoding of bytes. It has the advantages that the Unicode characters corresponding to the familiar ASCII set have the same byte values as ASCII, and that Unicode characters transformed into UTF-8 can be used with much existing software without extensive software rewrites.

UTF-16 is popular in many environments that need to balance efficient access to characters with economical use of storage. It is reasonably compact and all the heavily used characters fit into a single 16-bit code unit, while all other characters are accessible via pairs of 16-bit code units.

UTF-32 is popular where memory space is no concern, but fixed width, single code unit access to characters is desired. Each Unicode character is encoded in a single 32-bit code unit when using UTF-32.


All three encoding forms need at most 4 bytes (or 32-bits) of data for each character.

## Defining Elements of Text

Written languages are represented by textual elements that are used to create words and sentences. These elements may be letters such as "w" or "M"; characters such as those used in Japanese Hiragana to represent syllables; or ideographs such as those used in Chinese to represent full words or concepts.

The definition of *text elements* often changes depending on the process handling the text. For example, in historic Spanish language sorting, "ll"; counts as a single text element. However, when Spanish words are typed, "ll" is two separate text elements: "l" and "l".

To avoid deciding what is and is not a text element in different processes, the Unicode Standard defines *code elements* (commonly called "characters"). A code element is fundamental and useful for computer text processing. For the most part, code elements correspond to the most commonly used text elements. In the case of the Spanish "ll", the Unicode Standard defines each "l" as a separate code element. The task of

**Code Charts**

[Home](#) | [Site Map](#) | [Search](#)

**Related Links**  
[About These Charts](#)  
[Unicode Character Names Index](#)  
[Unihan Database](#)  
[Where is my Character?](#)  
[The Unicode Standard, Version 4.0](#)

































































**Additional Charts-Related Resources**  
[Normalization Charts](#)  
[Collation Charts](#)  
[Case Mapping Charts](#)  
[Script Name Charts](#)  
[Char Conversion Charts](#)  
[Javascript Unicode Charts](#)  
[Unibook Character Browser](#)




























































**Locating a Chart by Character Code**  
To find out the starting code point of any chart, let the mouse hover over the icon or the text. A popup with the starting code point will appear, or the status bar will show the address of the link. For example, the link for Basic Latin is U0000.pdf, indicating that the chart starts at U+0000.

**Viewing PDF Files**  
Viewing PDF files requires the free [Adobe Acrobat Reader](#). If you experience problems viewing one of the PDF files in your browser, try saving the file to your disk before opening it.

## Code Charts (PDF Version)

The charts in this list are arranged in code point order. For an **alphabetical** index of characters and blocks, use the [Unicode Character Names Index](#).

 <a href="#">Basic Latin</a>	 <a href="#">Geometric Shapes</a>
 <a href="#">Latin-1 Supplement</a>	 <a href="#">Miscellaneous Symbols</a>
 <a href="#">Latin Extended-A</a>	 <a href="#">Dingbats</a>
 <a href="#">Latin Extended-B</a>	 <a href="#">Miscellaneous Mathematical Symbols-A</a>
 <a href="#">IPA Extensions</a>	 <a href="#">Supplemental Arrows-A</a>
 <a href="#">Spacing Modifier Letters</a>	 <a href="#">Braille Patterns</a>
 <a href="#">Combining Diacritical Marks</a>	 <a href="#">Supplemental Arrows-B</a>
 <a href="#">Greek</a>	 <a href="#">Miscellaneous Mathematical Symbols-B</a>
 <a href="#">Cyrillic</a>	 <a href="#">Supplemental Mathematical Operators</a>
 <a href="#">Cyrillic Supplement</a>	 <a href="#">Miscellaneous Symbols and Arrows</a>
 <a href="#">Armenian</a>	 <a href="#">CJK Radicals Supplement</a>
 <a href="#">Hebrew</a>	 <a href="#">Kangxi Radicals</a>
 <a href="#">Arabic</a>	 <a href="#">Ideographic Description Characters</a>
 <a href="#">Syriac</a>	 <a href="#">CJK Symbols and Punctuation</a>
 <a href="#">Thaana</a>	 <a href="#">Hiragana</a>
 <a href="#">Devanagari</a>	 <a href="#">Katakana</a>
 <a href="#">Bengali</a>	 <a href="#">Bopomofo</a>
 <a href="#">Gurmukhi</a>	 <a href="#">Hangul Compatibility Jamo</a>
 <a href="#">Gujarati</a>	 <a href="#">Kanbun</a>
 <a href="#">Oriya</a>	 <a href="#">Bopomofo Extended</a>
 <a href="#">Tamil</a>	 <a href="#">Katakana Phonetic Extensions</a>
 <a href="#">Telugu</a>	 <a href="#">Enclosed CJK Letters and Months</a>
 <a href="#">Kannada</a>	 <a href="#">CJK Compatibility</a>
 <a href="#">Malayalam</a>	 <a href="#">CJK Unified Ideographs Extension A (1.5MB)</a>
 <a href="#">Sinhala</a>	 <a href="#">Yijing Hexagram Symbols</a>
 <a href="#">Thai</a>	 <a href="#">CJK Unified Ideographs (5MB)</a>
 <a href="#">Lao</a>	 <a href="#">Yi Syllables</a>
 <a href="#">Tibetan</a>	 <a href="#">Yi Radicals</a>
 <a href="#">Myanmar</a>	 <a href="#">Hangul Syllables (7MB)</a>
 <a href="#">Georgian</a>	 <a href="#">High Surrogates</a>
 <a href="#">Hangul Jamo</a>	 <a href="#">Low Surrogates</a>
 <a href="#">Ethiopic</a>	 <a href="#">Private Use Area</a>

 <a href="#">Cherokee</a>	 <a href="#">CJK Compatibility Ideographs</a>
 <a href="#">Unified Canadian Aboriginal Syllabic</a>	 <a href="#">Alphabetic Presentation Forms</a>
 <a href="#">Ogham</a>	 <a href="#">Arabic Presentation Forms-A</a>
 <a href="#">Runic</a>	 <a href="#">Variation Selectors</a>
 <a href="#">Tagalog</a>	 <a href="#">Combining Half Marks</a>
 <a href="#">Hanunoo</a>	 <a href="#">CJK Compatibility Forms</a>
 <a href="#">Buhid</a>	 <a href="#">Small Form Variants</a>
 <a href="#">Tagbanwa</a>	 <a href="#">Arabic Presentation Forms-B</a>
 <a href="#">Khmer</a>	 <a href="#">Halfwidth and Fullwidth Forms</a>
 <a href="#">Mongolian</a>	 <a href="#">Specials</a>
 <a href="#">Limbu</a>	 <a href="#">Linear B Syllabary</a>
 <a href="#">Tai Le</a>	 <a href="#">Linear B Ideograms</a>
 <a href="#">Khmer Symbols</a>	 <a href="#">Aegean Numbers</a>
 <a href="#">Phonetic Extensions</a>	 <a href="#">Old Italic</a>
 <a href="#">Latin Extended Additional</a>	 <a href="#">Gothic</a>
 <a href="#">Greek Extended</a>	 <a href="#">Ugaritic</a>
 <a href="#">General Punctuation</a>	 <a href="#">Deseret</a>
 <a href="#">Superscripts and Subscripts</a>	 <a href="#">Shavian</a>
 <a href="#">Currency Symbols</a>	 <a href="#">Osmanya</a>
 <a href="#">Combining Marks for Symbols</a>	 <a href="#">Cypriot Syllabary</a>
 <a href="#">Letterlike Symbols</a>	 <a href="#">Byzantine Musical Symbols</a>
 <a href="#">Number Forms</a>	 <a href="#">Musical Symbols</a>
 <a href="#">Arrows</a>	 <a href="#">Tai Xuan Jing Symbols</a>
 <a href="#">Mathematical Operators</a>	 <a href="#">Mathematical Alphanumeric Symbols</a>
 <a href="#">Miscellaneous Technical</a>	 <a href="#">CJK Unified Ideographs Extension B (13MB)</a>
 <a href="#">Control Pictures</a>	 <a href="#">CJK Compatibility Ideographs Supplement</a>
 <a href="#">Optical Character Recognition</a>	 <a href="#">Tags</a>
 <a href="#">Enclosed Alphanumerics</a>	 <a href="#">Variation Selectors Supplement</a>
 <a href="#">Box Drawing</a>	 <a href="#">Supplementary Private Use Area-A</a>
 <a href="#">Block Elements</a>	 <a href="#">Supplementary Private Use Area-B</a>

The character code charts were last updated for post-Unicode 4.0 errata

Copyright © 1991-2003 Unicode, Inc.  
All Rights Reserved  
[Terms of Use](#)

Last updated: - domingo, 31 de Agosto de 2003 20:40:21

# How Saving A Bit Map Works

**B**it-mapped graphics represent one of the most popular graphics file formats used on PCs. For example, the Paint program in Windows creates bit-mapped images. Each bit map includes one or more bits that describe the color of each tiny square, known as a **pixel**, that form the image.

In a 24-bit color graphic, such as the one below, each pixel is represented by 3 bytes: one red, one green, and one blue. These are known as the graphic's **RGB values**. Because this is a 24-bit graphic, each of its pixels has the possibility of using 256 shades of red, green, and blue. This means a pixel could be any one of more than 16 million colors.

To save these files, the RGB values must be converted into palette values, which are determined by the program you are using to save the file. Each pixel in the file is listed by its palette



number rather than by each color value number. The palette numbers are used to limit the amount of bits needed to describe each pixel. Although bit-mapped files can still be quite large, saving the values for each pixel takes only a couple of seconds. Here is how it works:



You can create a 24-bit image in a graphics program such as Paint.



A graphics program saves the image line by line, from the bottom to the top.



Each of the pixel's three-color values, RGB (red-green-blue), are read from left to right.

R 250	R 244	R 238
G 210	G 195	G 182
B 94	B 69	B 51
R 242	R 235	R 222
G 190	G 176	G 160
B 60	B 42	B 26
R 228	R 218	R 201
G 167	G 153	G 148
B 27	B 17	B 53

A graphics program translates the RGB values into palette values. The palette values are a software-specific decision; each program's values are different.

FAD25E	F4C345	EEB633
F2BE3C	EBB02A	DEA01A
E4A71B	DA9911	C99435

Each palette value, a hexadecimal value in this case, is stored in the same order as displayed in the image.



The pixel values are stored in the bit-mapped file in the same width and depth as the original image.

## Forming A Pixel

A pixel is the smallest part of an image that a computer's monitor can control. Each pixel consists of three colors: red, green, and blue. Each of the three colors is assigned a value that shows its intensity; the values are from 0 to 255. You can think of each value as a percentage. For example, 127 has a 50% intensity. These are known as the RGB values.



Compiled by Kyle Schurman  
Graphics & Design by Lori Garriss



Part1 - [Part2](#) - [Single Page](#)

Top Document: [JPEG image compression FAQ, part 1/2](#)

Previous Document: [News Headers](#)

Next Document: [\[2\] Why use JPEG?](#)

---

## [1] What is JPEG?

---

JPEG (pronounced "jay-peg") is a standardized image compression mechanism. JPEG stands for Joint Photographic Experts Group, the original name of the committee that wrote the standard.

JPEG is designed for compressing either full-color or gray-scale images of natural, real-world scenes. It works well on photographs, naturalistic artwork, and similar material; not so well on lettering, simple cartoons, or line drawings. JPEG handles only still images, but there is a related standard called MPEG for motion pictures.

JPEG is "lossy," meaning that the decompressed image isn't quite the same as the one you started with. (There are lossless image compression algorithms, but JPEG achieves much greater compression than is possible with lossless methods.) JPEG is designed to exploit known limitations of the human eye, notably the fact that small color changes are perceived less accurately than small changes in brightness. Thus, JPEG is intended for compressing images that will be looked at by humans. If you plan to machine-analyze your images, the small errors introduced by JPEG may be a problem for you, even if they are invisible to the eye.

A useful property of JPEG is that the degree of lossiness can be varied by adjusting compression parameters. This means that the image maker can trade off file size against output image quality. You can make \*extremely\* small files if you don't mind poor quality; this is useful for applications such as indexing image archives. Conversely, if you aren't happy with the output quality at the default compression setting, you can jack up the quality until you are satisfied, and accept lesser compression.

Another important aspect of JPEG is that decoders can trade off decoding speed against image quality, by using fast but inaccurate approximations to the required calculations. Some viewers obtain remarkable speedups in this way. (Encoders can also trade accuracy for speed, but there's usually less reason to make such a sacrifice when writing a file.)

---

## [2] Why use JPEG?

---

There are two good reasons: to make your image files smaller, and to store 24-bit-per-pixel color data instead of 8-bit-per-pixel data.

Making image files smaller is a win for transmitting files across networks and for archiving libraries of images. Being able to compress a 2 Mbyte

full-color file down to, say, 100 Kbytes makes a big difference in disk space and transmission time! And JPEG can easily provide 20:1 compression of full-color data. If you are comparing GIF and JPEG, the size ratio is usually more like 4:1 (see "[4] How well does JPEG compress images?").

Now, it takes longer to decode and view a JPEG image than to view an image of a simpler format such as GIF. Thus using JPEG is essentially a time/space tradeoff: you give up some time in order to store or transmit an image more cheaply. But it's worth noting that when network transmission is involved, the time savings from transferring a shorter file can be greater than the time needed to decompress the file.

The second fundamental advantage of JPEG is that it stores full color information: 24 bits/pixel (16 million colors). GIF, the other image format widely used on the net, can only store 8 bits/pixel (256 or fewer colors). GIF is reasonably well matched to inexpensive computer displays --- most run-of-the-mill PCs can't display more than 256 distinct colors at once. But full-color hardware is getting cheaper all the time, and JPEG photos look \*much\* better than GIFs on such hardware. Within a couple of years, GIF will probably seem as obsolete as black-and-white MacPaint format does today. Furthermore, JPEG is far more useful than GIF for exchanging images among people with widely varying display hardware, because it avoids prejudging how many colors to use (see "[8] What is color quantization?"). Hence JPEG is considerably more appropriate than GIF for use as a Usenet and World Wide Web standard photo format.

A lot of people are scared off by the term "lossy compression". But when it comes to representing real-world scenes, \*no\* digital image format can retain all the information that impinges on your eyeball. By comparison with the real-world scene, JPEG loses far less information than GIF. The real disadvantage of lossy compression is that if you repeatedly compress and decompress an image, you lose a little more quality each time (see "[10] Does loss accumulate with repeated compression/decompression?"). This is a serious objection for some applications but matters not at all for many others.

---

## [3] When should I use JPEG, and when should I stick with GIF?

---

JPEG is \*not\* going to displace GIF entirely; for some types of images, GIF is superior in image quality, file size, or both. One of the first things to learn about JPEG is which kinds of images to apply it to.

Generally speaking, JPEG is superior to GIF for storing full-color or gray-scale images of "realistic" scenes; that means scanned photographs, continuous-tone artwork, and similar material. Any smooth variation in color, such as occurs in highlighted or shaded areas, will be represented more faithfully and in less space by JPEG than by GIF.

GIF does significantly better on images with only a few distinct colors, such as line drawings and simple cartoons. Not only is GIF lossless for such images, but it often compresses them more than JPEG can. For example, large areas of pixels that are all \*exactly\* the same color are compressed very efficiently indeed by GIF. JPEG can't squeeze such data as much as GIF does without introducing visible defects. (One implication of this is that large single-color borders are quite cheap in GIF files, while they are best avoided in JPEG files.)

Computer-drawn images, such as ray-traced scenes, usually fall between photographs and cartoons in terms of complexity. The more complex and subtly rendered the image, the more likely that JPEG will do well on it. The same goes for semi-realistic artwork (fantasy drawings and such). But icons that use only a few colors are handled better by GIF.

JPEG has a hard time with very sharp edges: a row of pure-black pixels adjacent to a row of pure-white pixels, for example. Sharp edges tend to come out blurred unless you use a very high quality setting. Edges this sharp are rare in scanned photographs, but are fairly common in GIF files: consider borders, overlaid text, etc. The blurriness is particularly objectionable with text that's only a few pixels high. If you have a GIF with a lot of small-size overlaid text, don't JPEG it. (If you want to attach descriptive text to a JPEG image, put it in as a comment rather than trying to overlay it on the image. Most recent JPEG software can deal with textual comments in a JPEG file, although older viewers may just ignore the comments.)

Plain black-and-white (two level) images should never be converted to JPEG; they violate all of the conditions given above. You need at least about 16 gray levels before JPEG is useful for gray-scale images. It should also be noted that GIF is lossless for gray-scale images of up to 256 levels, while JPEG is not.

If you have a large library of GIF images, you may want to save space by converting the GIFs to JPEG. This is trickier than it may seem --- even when the GIFs contain photographic images, they are actually very poor source material for JPEG, because the images have been color-reduced. Non-photographic images should generally be left in GIF form. Good-quality photographic GIFs can often be converted with no visible quality loss, but only if you know what you are doing and you take the time to work on each image individually. Otherwise you're likely to lose a lot of image quality or waste a lot of disk space ... quite possibly both. Read sections 8 and 9 if you want to convert GIFs to JPEG.

---

## [4] How well does JPEG compress images?

---

Very well indeed, when working with its intended type of image (photographs and suchlike). For full-color images, the uncompressed data is normally 24 bits/pixel. The best known lossless compression methods can compress such data about 2:1 on average. JPEG can typically achieve 10:1 to 20:1 compression without visible loss, bringing the effective storage requirement down to 1 to 2 bits/pixel. 30:1 to 50:1 compression is possible with small to moderate defects, while for very-low-quality purposes such as previews or archive indexes, 100:1 compression is quite feasible. An image compressed 100:1 with JPEG takes up the same space as a full-color one-tenth-scale thumbnail image, yet it retains much more detail than such a thumbnail.

For comparison, a GIF version of the same image would start out by sacrificing most of the color information to reduce the image to 256 colors (8 bits/pixel). This provides 3:1 compression. GIF has additional "LZW" compression built in, but LZW doesn't work very well on typical photographic data; at most you may get 5:1 compression overall, and it's not at all uncommon for LZW to be a net loss (i.e., less than 3:1 overall compression). LZW \*does\* work well on simpler images such as line drawings, which is why

GIF handles that sort of image so well. When a JPEG file is made from full-color photographic data, using a quality setting just high enough to prevent visible loss, the JPEG will typically be a factor of four or five smaller than a GIF file made from the same data.

Gray-scale images do not compress by such large factors. Because the human eye is much more sensitive to brightness variations than to hue variations, JPEG can compress hue data more heavily than brightness (gray-scale) data. A gray-scale JPEG file is generally only about 10%-25% smaller than a full-color JPEG file of similar visual quality. But the uncompressed gray-scale data is only 8 bits/pixel, or one-third the size of the color data, so the calculated compression ratio is much lower. The threshold of visible loss is often around 5:1 compression for gray-scale images.

The exact threshold at which errors become visible depends on your viewing conditions. The smaller an individual pixel, the harder it is to see an error; so errors are more visible on a computer screen (at 70 or so dots/inch) than on a high-quality color printout (300 or more dots/inch). Thus a higher-resolution image can tolerate more compression ... which is fortunate considering it's much bigger to start with. The compression ratios quoted above are typical for screen viewing. Also note that the threshold of visible error varies considerably across images.

---

Top Document: JPEG image compression FAQ, part 1/2  
Next Document: [5] What are good "quality" settings for JPEG?

Part I - [Part2 - Single Page](#)

---

*Send corrections/additions to the FAQ Maintainer:*  
[jpeg-info@uunet.uu.net](mailto:jpeg-info@uunet.uu.net)

Last Update September 24 2003 @ 00:30 AM



# CompuServe GIF Image Format

*Contributed by Steve Neuendorffer*

One of the widely used formats for representing images on the Internet is CompuServe's GIF, or graphics interchange format. An image file, like any other computer file, is a collection of bits. If you attempt to read the file in a text editor (such as emacs, which can open binary files) you will see something like this:

```
GIF89at+ÿJ´ ¨Ñ£H*]Ê´©Ó$P£JµªÖ«X³jÝÊµ«¨´ÄK¶-û³
```

followed by several hundred or thousand more lines of absolutely incomprehensible nonsense, and an occasional beep. How does this represent an image?

The text editor is expecting a binary sequence representing characters in the ASCII format. The American Standard Code for Information Interchange. In ASCII, each 8 bits are grouped together in a byte, and each byte corresponds to a single character. This allows for 2^8=256 possible characters. Since there are only 26 letters in the English alphabet, plus 26 upper case letters, plus 10 digits, plus a dozen or so punctuation marks, there are quite a few extra characters leftover. Your text editor interprets these additional characters as shown above. If you are using a text editor that supports 16-bit unicode characters, you might see instead Chinese or Hebrew characters, for example. A few of the characters are interpreted as control characters, like the Carriage Return (ASCII 13), Line Feed (ASCII 10), and Bell (ASCII 7), which would ring a physical bell on old Teletype machines. Modern computers usually beep whenever a Bell character comes along, hence the beeps that the text editor produces.

Pictures in a computer are divided into pixels (picture elements), organized horizontally and vertically in lines, much like lines of characters on a page. The GIF must represent a rather large number of pixels efficiently, or the file size (and Internet transport time) gets too large.

In a GIF image, the first pixel in the file goes in the upper left hand corner, and the second one goes just to its right. The image is scanned from left to right, top to bottom, until all the pixels have been specified. (This is not the only way of ordering pixels in a computer. TIFF images happen to start at the bottom and work their way up.)

Each pixel has a color. In GIF images, a color is specified using Red, Green, and Blue components. With 8 bits for each component, there are over 2^24, or about 16 million possible colors. A naive representation of the image would simply store three bytes for each pixel. But then a 640x480 pixel image (which is a modest size) would be around a megabyte of data. At a modem speed of 56 kilobits per second, it would take 131 seconds, more than two minutes, to download the image. GIF compresses the data, reducing the number of bits to represent the image.

The first kind of compression that GIF uses is called a colormap. Instead of allowing the image to contain all 16 million colors, GIF restricts the image to a maximum of, say, 256 out of the 16 million (the number of colors in the colormap can be varied). It can be any 256 out of the 16 million, so there is no loss of richness of possible colors. But no more than 256 distinct colors can be used simultaneously in any one image. The colors are stored in a colormap table, and the color for each pixel is specified as an index into the table. So instead of using 24 bits for each pixel, a file only contains an 8 bit index. (A 24-bit display of a modern computer can display all 16 million colors simultaneously, so multiple GIF images with different colormap tables can be simultaneously displayed with good color fidelity.)

With colormap table of 256 entries, the above scheme reduces the amount of data by a factor of three. But GIF does better than this. The second kind of compression that GIF includes is called run-length coding. This makes use of the fact that neighboring pixels are often the same color in a typical image. When several pixels have the same color, instead of storing them individually, they are stored as a run length followed by the color. For example, a sequence of three Blue Blue Blue" or "3, Blue". The specifics of how this is done is a little complex; GIF uses a sophisticated variation of run-length coding known as Lempel-Ziv-Welch coding. The algorithm is also used in several file-compression utilities as well. See the details in [an article on GIF compression](#).)

How is a GIF file converted into an image? To make things easier, here are the first few bytes of the file above written in binary (each line is a byte, or 8 bits):

```
1) 01000111 -> 'G'
2) 01001001 -> 'I'
3) 01000110 -> 'F'
4) 00111001 -> '8'
```

```
5) 00111010 -> '9'
6) 01100001 -> 'a'
7) 01110100
8) 00000011 -> 884 pixels wide
9) 00101011
10) 00000100 -> 1067 pixels high
11) 11110111 -> Global colormap present, 256 colors, 24 bits per color.
12) 11111111 -> background is color index 255.
13) 00000000 -> zero
14) 00000000
15) 00000000
16) 00000000 -> color 0 in the colormap = (0, 0, 0) = black
17) 10000000
18) 00000000
19) 00000000 -> color 1 in the colormap = (128, 0, 0) = red
20) 00000000
21) 10000000
22) 00000000 -> color 2 in the colormap = (0, 128, 0) = green
```

The first six bytes contain six ASCII characters. For the file above the first six characters are the ASCII characters "GIF89a", indicating that this file is compatible with the GIF standard written in 1989. You may also see "GIF87a" which was an earlier version of the standard. There is no information there about any pixels, just information to the decoder about what kind of file it is looking at.

From here on, the bytes do not correspond to ASCII characters. Next comes a sequence of bytes that give the size of the screen, two bytes for the width and two bytes for the height. The image above happens to be 884 pixels by 1067 pixels.

Byte 11 tells how the colors are stored in the file. In the file above, and in most GIF files, this byte indicates that the file contains its own colormap (as opposed to using the default colormap), that the colormap contains 255 entries, and that each color in the color map is specified using 24 bits. If the colormap is not present, then the GIF standard assumes that a pre-defined default colormap is used.

Byte 12 gives the index of the background color in the colormap, in this case it is color 255. Byte 13 is always zero.

The first 13 bytes are the "screen descriptor". In the file above, this is followed by 768 bytes of colormap information. In GIF files, the red component comes first, followed by green and then blue. 0 indicates the minimum intensity of that component, while 255 indicates the maximum intensity.

Following the global colormap is an "image descriptor." A GIF file can contain more than one image. Each image can be individually positioned on the screen and can even have its own colormap. The file we are using just has one image with the following descriptor:

```
1) 00101100 -> ",",
2) 00000000
3) 00000000 -> 0 pixels offset to the right.
4) 00000000
5) 00000000 -> 0 pixels offset to the bottom.
6) 01110100
7) 00000011 -> 884 pixels wide
8) 00101011
9) 00000100 -> 1067 pixels high
10) 00000000 -> No local colormap, non-interlaced.
```

The first character in the image descriptor is always the ASCII code for a comma. The next 8 bytes give the position and size of the image, which in this case starts in the upper left of the screen, and is the same size as the screen. Byte ten specifies that there is no local colormap, and the lines of the image are scanned in normal order. (GIF also allows images to be scanned in an interleaved fashion, which permits a browser to display a blurry approximation of the image before it has received all the data. For more information about this, see the [GIF87a specification](#).)

Finally we get to the image data for the pixels, run-length coded. After all the image data, there is a single ASCII semicolon, binary code 00111011.

# stl.caltech.edu - Introduction to PNG

*IMPERFECT SYSTEMS INFURIATE HACKERS, WHOSE PRIMAL INSTINCT IS TO DEBUG THEM*

**De Novo:** | [Index & Plan](#) | [Links](#) | [Culture](#) | [Parrises Squares](#) | [GeForce FX](#) |  
**Reviews:** | [Rating System](#) |

**Primary:** | [bwtzip](#) | [The Quotation Collection](#) | [Downloads](#) | [Anime/SF](#) | [Deus Ex](#) |  
**Diversions:** | [Coloring](#) | [Paper Airplane](#) | [Origami Polyhedra](#) | [Foundation](#) |  
**Site Info:** | [Archived News](#) | [Personal Page](#) |  
**Essays:** | [Space](#) | [Internet](#) | [Programming](#) | [C Intro](#) | [Games](#) | [Bits & Bytes](#) |  
**Tech:** | [C++](#) | [Computer](#) | [Random Work](#) | PNG |  
**Random:** | [Wallpaper](#) | [Diet](#) | [Mersenne Primes](#) | [Book Reviews](#) | [pngacc](#) | [LASTLY](#) |

## PNG introduction:

PNG is a way to store image files. As such, it's a tool. However, it's an extremely useful, versatile, and efficient tool; once I became acquainted with it, I immediately realized that it was beautiful and I haven't looked back since I've started using it. This page is meant to be a simple, conceptual introduction to the PNG file format, intended somewhat for those who will see it on the Internet, but **this page is mostly directed towards those who will use the PNG format casually**. For example, someone who maintains a small set of web pages (or even a large set!) and needs to save images used on the site. The only people this is not directed to are the deeply insane programmers who want to work with PNG on a detailed level right down to the bits (ahem); those people will probably want to look at [The PNG Home Page](#) and read the specifications there. In fact, I recommend that even casual users take a look at the PNG Home Page once they're done with this page; the specification and related pages go into deep detail that I need not copy; the only thing that the PNG Home Page lacks is a good introduction for casual users. Hence this page.

## Preliminaries:

PNG stands for Portable Network Graphics (unofficially it's PNG's Not Gif). It is always pronounced "ping". PNG is a true lossless format, unlike JPEG. I won't get into the lossless/lossy difference here, as this page is meant to show PNG's useful features and is not a guide as to when PNG is better than JPEG. (Okay, quickly: use JPEG when absolute accuracy is not necessary and natural scenes are photographed; use PNG everywhere else. Everywhere.) PNG is a replacement for GIF, **and more**, in that it's technically superior to GIF in all areas except for one (animation, but Multiple-Image Network Graphics will kill GIF eventually). Every area in which PNG beats GIF is another reason to use and want PNG instead of GIF.

## New vs. old:

**PNG is not a new thing!** It was completely frozen as a standard all the way back in 1995. Yes, that's back in the 90's. (GIF was completed in the late 80's.) Incidentally, Windows 95 came out in, um, 1995; does anyone consider IT to be a newfangled thing that no one ought to learn? And the SHA-1 cryptographic hash algorithm was

created in 1995 as well; it's the gold standard of hash algorithms as of 2000, completely superior to the older MD5 and MD4 algorithms. Five years is plenty of time. The reason that people seem to think that PNG is newfangled and not broadly supported is that they don't realize how many benefits it offers (which this page attempts to combat) and that Internet browsers, even in 2000, have haphazard support for this well-established standard. (They DO support it enough to be broadly useful, though, which is another myth! 8-bit PNGs, optionally with on-off transparency, are supported in one fashion or another in almost all modern browsers, which is enough for most Web uses, even though it doesn't begin to unlock all of PNG's advantages. Sometimes variable transparency is broken, sometimes not.) If you're worried about browser support, look at it another way: you interact with images on the Internet all the time. Should web pages not use images because of text-based browsers like Lynx that can't view the images? NO! Web pages certainly SHOULD use images when necessary, because there's no reason to be stuck in the past. (Adding ALT tags to the images is a common courtesy to blind people, people with Lynx, or people with images disabled who want faster HTML viewing.) Same thing with PNG. Go ahead and use PNG, and include an ALT tag for the people who are still using non-PNG-compliant browsers. The more sites that use PNG, the more pressure on browser makers to include full nonbroken support. And even if you don't intend to create PNG images, once you've seen the benefits of PNG on this page, start to demand PNG from sites that still use GIF images. **Become a PNG advocate and make the world a better place.**

## Why PNG matters to you:

There's a long list of PNG features on the PNG home page. However, only a few of them really matter to casual users. Here are the important ones that matter, in decreasing order of importance:

- PNG's compression, properly done, is **significantly better** (20% better, usually) than GIF's compression. Result: faster-loading images! (This is a big reason for pure users to demand PNG from web sites. The benefits to site maintainers should be obvious as well.)
- PNG images can be interlaced, which makes them appear **faster and better-looking** across modem connections. Additionally, PNG interlacing is infinitely better than GIF interlacing. See below for a real graphical demonstration of this statement.
- PNG supports a large but carefully-chosen range of bit depths, from monochrome all the way to 64-bit color (yes, 64-bit). Thus it is useful not only on the Internet but anywhere else a lossless format is needed.
- Because of broken browsers, this isn't significant now, but will be when more browsers follow Mac-IE5's example: PNG supports variable transparency, so small buttons and the like can be displayed on **any background**, dark or light, and look good.



- And PNG supports robust error detection, gamma correction, text chunks, and a ton of other nifty stuff, none of which users ought to worry about because it's done behind their backs.

The better compression and turbo-nifty interlacing should be significant enough to entice users to switch from preferring GIF to preferring PNG, and is often enough. But here's a bonus: there's an ideological reason to prefer PNG. PNG is not encumbered by any patents at all: J. Random Hacker can make whatever programs he wants to that deal with PNG without answering to anyone. GIF is encumbered by a foggy patent status, **in addition to being vastly technically inferior**. This ideological reason may be important to you if you've heard of "free software" before, otherwise, it probably doesn't matter to you.

What **should** matter is that you wouldn't stand a monitor that could only display 256 colors, and you don't need to stand a graphics format (GIF) that can only display 256 colors and one bit of transparency. **It's time to move on to better things**. It's time to move on to PNG.

### Interlacing:

There are four ways to transmit an image over the Internet; over a fast connection there won't be any apparent difference, but over a modem connection the difference is stunningly obvious. Choosing the right way can make your connection seem much faster than it really is.

The first, and stupid, method is to wait until every bit of image data has been sucked through the modem before displaying the whole image. This is so blindingly dumb that not even Internet Explorer does it.

A better method is to display image data as it is received, resulting in a top-down filling in of the image. One variant (the one that everyone has seen) of JPEG does this. This is noninterlaced display, and both GIF and PNG are capable of it as well. Non-interlaced images are smaller than interlaced images.

**However**, while interlacing expands file size, it makes images MUCH more easily viewable, a benefit that well outweighs the slight filesize cost. There are two methods of interlacing that are popular: GIF interlacing and PNG interlacing (the latter is also called "Adam7"). GIF interlacing is remarkably stupid; PNG interlacing remarkably cool. GIFs, when interlaced, use a **one-dimensional** scheme: every eighth horizontal line is transmitted in the first "pass", filling up the dimensions of the image in 1/8th of the time that the entire image will take to download. The next pass transmits every fourth line, making the image less distorted. The next pass transmits every second line, making the image even less distorted, and the fourth and final pass transmits the remaining lines.

However, GIF interlacing is STUPID in that it's one-dimensional: after the first pass, the image is stretched by a factor of **eight to one**. (You'll see this shortly below.) PNG uses a **two-dimensional** scheme. Don't let the details concern you; what

matters is that instead of four passes through the image, PNG makes seven passes. In 1/64 of the time that the whole image will take to display, already one pass is completed, showing the image in a very rough manner. Successive passes fill in more information, never distorting the pixels by more than a factor of two to one. And interestingly, the browser/viewer can **interpolate** the incoming PNG as it is received, producing stunningly beautiful results. (To interpolate means to predict pixels in between given pixels. If only a small fraction of pixels have been downloaded, we may want to fill in the rest of the image.) The "fading-in, focusing-in" effect produced by an interpolated, interlaced PNG is nifty to watch in its own right.

### An interlacing demonstration:

Let's say, for the sake of argument, that you are downloading a large image. Let's walk through how this would look if you were looking at an interlaced and bicubic interpolated PNG, an interlaced and bilinear interpolated PNG, an interlaced and noninterpolated PNG, or an interlaced GIF. (Bicubic interpolation looks better than bilinear interpolation: the algorithm used is better.) Let me say something first that is sort of complicated, before we get to the pretty pictures:

#### Technical note:

The original source picture was taken with a very nice digital camera and started life as a 24-bit 3.3 megapixel image. I reduced and cropped it from its original 3.3-megapixel size to make it 256x256 pixels so that this page wouldn't actually take forever to load, and so that it would (mostly) fit on your screen. Then I produced this demonstration. No GIFs were actually involved here, but I know how to exactly simulate GIF interlacing. The following pictures are all 24-bit, and I am using PNG files to show them to you. What you are seeing is what you would actually see if you were downloading a 24-bit PNG: it would be interpolated with 24-bit smoothness. (Even if you were downloading an 8-bit PNG, it still could be interpolated with 24-bit smoothness.) It is a small cheat to do this in photo editors, but an appropriately programmed browser could do the same thing, even in the middle of a pass (mostly). Now, as for the "GIF", I am cheating by making it a 24-bit image. In reality, GIFs suck and can only be 8-bit. In this case, the 8-bit version of the image is very nearly indistinguishable from the original. (For larger images, you cannot get away with using 8-bit, as some colors must be noticeably corrupted.) In any case, bit-for-bit, a PNG image can either store more pixels or more bit depth than a similarly sized GIF, because of the difference in compression. Bear with me, I am **only trying to show you the difference between various interlacing and interpolation schemes**. I am using a 24-bit image for the "GIF" so that any differences you see are due to interlacing and interpolation; if I used an 8-bit image to simulate the GIF, I'd still have to make a similar and long explanation. Don't worry if that made no sense, because I'm fairly confident you won't care in the first place. In fact, if you didn't fully understand what I said, **ignore it**, because it's only information about how I created this demonstration and does not affect the point I'm making; namely, that PNG interlacing is vastly superior to GIF interlacing. This explanation is here so that the curious can convince themselves that everything I am doing is proper and (not much) fakery is involved. If you are curious, to make the following images I used resizing in MS Photo Editor to throw out pixels (I used a nearest neighbor method, so I am actually throwing out pixels, and not doing a bilinear interpolation down). Then I used MS Photo Editor and resizing larger (no smoothing: nearest neighbor) to simulate the GIF and noninterpolated PNG. I used MS Photo Editor and resizing with smoothing (bilinear interpolation) to simulate the bilinear interpolated PNG. I used Adobe Photoshop 5.5 and bicubic resizing to create the final image. If you are wondering, a browser could also perform interpolation on a downloading GIF, but it wouldn't help the fact that GIF's interlacing scheme sucks.

Here's what's important to remember:

**In the following pictures, we are pretending that we are seeing an 8-bit PNG and an 8-bit GIF downloaded, both at the same rate. We are looking at three**

## Web Graphics



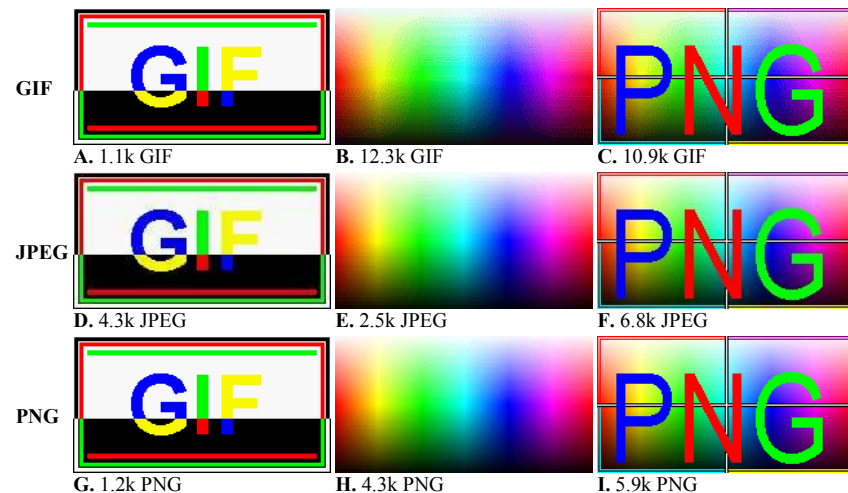
There are many web pages devoted to describing the relative merits of using JPEG, GIF and PNG images. This page is a short summary of some of the things I have learned about these formats while designing the web pages for my department and while writing software to view medical images. This is not meant as a definitive guide, more a basic introduction with a few useful links.

There are three common formats used by web pages: GIF, JPEG and PNG. Each of these formats uses compression techniques to reduce the amount of time required to download images. In general, each has its own strengths and weaknesses:

Format	Strength	Weakness
GIF	Lossless (if < 257 colours)	Maximum of 256 colours, poor compression in situations where there are gradual changes in colours. <a href="#">Patented technology, requires a license. This patent will expire on 20 June 2003.</a>
JPEG	Millions of colours	Lossy: sharp edges will appear blurred. Does not support transparency.
PNG	Millions of colors, lossless	Not supported by older browsers.

Below I have presented three images that conform to the strengths of each format.

1. The image in the leftmost column uses a few colors and the same color is present in large chunks of the graphic. This first image (and most line art) is well suited for the GIF format. Note that the JPEG version (D) of this image requires more disk space and the edges of the colors look very blurry.
2. The middle image presents thousands of colours that gradually change across the image. This image (and most photographs) looks great when saved as a JPEG image. The GIF rendering of this image (B) requires more disk space and looks grainy (as more than 256 colours are in the image).
3. The final picture combines elements of the two other images: the background uses thousands of colours with gradual changes (which will hurt GIF compression), while the text has sharp, high contrast edges (that will hurt JPEG compression). This image demonstrates PNG's talent as a great all-round image format. Note that uncompressed, these images require around 62 k (when saved with 24-bits per pixel).

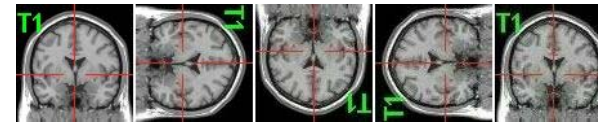


PNG images always look great and they usually achieve very respectable compression ratios. PNG protects you from the reduction in image quality found with JPEG (and images with more than 256 colours saved to the GIF format).

Unfortunately, PNG is not supported by [older web browsers](#). I am very wary of web sites that mention "best viewed with Netscape version 4 or later" or "best viewed with Internet Explorer version 4+". It seems to me that the idea of web pages is distributing information to others regardless of their hardware and software. For the moment, I use PNG for storing images on my hard disk, but stick to GIF and JPEG for my web pages.

An important consideration with JPEG images is to avoid unnecessary abrupt changes in the image. For example, many people like to add coloured frames to the borders of their images. These abrupt changes can greatly increase the file size and cause a number of artifacts in a JPEG image (as image F above demonstrates).

When using JPEG images, bear in mind that this is a lossy technique, and everytime you save a JPEG image, you will lose some of the quality. To demonstrate this, examine the image below. The original JPEG image is shown on the left. I rotated this image 90-degrees and saved the image four times. Note that the final image (shown on the far right) has degraded considerably: in particular the high frequency information of the green text and the red line have become blurry. This is a very common problem - especially with people who convert JPEG images between landscape and portrait format. Some software (including the excellent and free [IrfanView](#)) include functions to losslessly rotate JPEG images by 90-degree steps. In addition to rotations, JPEGs will also accumulate errors after repeated cropping, editing and other modifications.



This web page only describes the three graphics formats that are common on the web. I did not discuss many other common graphics formats (TIFF, BMP, etc). I have also described JPEG as a single process - in fact there are many ways of encoding JPEG images that can greatly influence the quality and compression ratios of the images. Likewise, there are various ways of encoding PNG images that can greatly modify the compression ratio. In the examples above, I used the standard 'baseline' settings for the JPEG and PNG images.

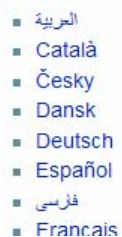
### Useful links:

- The [Yale web style guide](#) covers graphics and web design. They have an outstanding [GIF and JPEG comparison](#).
- Stanford's academic computing group provides a useful [GIF vs JPEG page](#) with a number of good links.
- [Lee Daniels](#) has a nice comparison of GIF and JPEG formats for web page graphics. He also has a number of useful [links](#).
- Metropolitan Community College describes the relative merits of [JPEG, GIF and PNG](#).
- A nice [comparison of JPEG, GIF and PNG](#) from Los Alamos Labs.
- Paul Bourke has created a huge number of web pages that are informative, pretty, and download quickly. Some of his useful graphics pages include an [introduction to bitmaps](#), a technical description of the [GIF format](#), suggestions for creating [good web pages](#), and tips for [tiling images](#) on web pages.

### Technical Links:

- [Mark Nelson's Data Compression library](#) contains many useful links - including JPEG, lossless JPEG, PNG and GIF.
- [Simon Fraser University's CMPT 365 course](#) has an excellent introduction to lossy and lossless JPEG images. In addition, there is a nice introduction to [lossless compression](#).
- The original specifications of the [JPEG](#) format are available as a [PDF](#) document. Useful for determined readers, but it is fairly technical.
- [Chris Watts](#) has written an easy to read introduction to the JPEG format. It has a nice description of the Huffman encoding algorithm.
- [Cristi Cuturicu](#) has also written an easy introduction to JPEG encoding.
- A simple listing of JPEG headers was compiled by [Oliver Fromme](#).
- [Chua Lye Heng](#) has written a description of different compression methods - perhaps the nicest description of Huffman encoding I have seen.
- The [PNG home](#) lists documentation, libraries and programs that support the PNG format.
- [Earl F Glynn's Graphics formats page](#) - this includes general information and information specifically about the Delphi programming language. He has a nice [page and free software](#) that demonstrates JPEG image quality and compression. His [color page](#) is a nice introduction to different color spaces.





- **MPEG-A:** Multimedia application format.
- **MPEG-B:** MPEG systems technologies.
- **MPEG-C:** MPEG video technologies.
- **MPEG-D:** MPEG audio technologies.
- **MPEG-E:** Multimedia Middleware.

