CHAPTER 27

Data Compression
Сжатие Данных

Data transmission and storage cost money. The more information being dealt with, the more it costs. In spite of this, most digital data are not stored in the most compact form. Rather, they are stored in whatever way makes them easiest to use, such as: ASCII text from word processors, binary code that can be executed on a computer, individual samples from a data acquisition system, etc. Typically, these easy-to-use encoding methods require data files about twice as large as actually needed to represent the information. Data compression is the general term for the various algorithms and programs developed to address this problem. A *compression program* is used to convert data from an easy-to-use format to one optimized for compactness. Likewise, an *uncompression program* returns the information to its original form. We examine five techniques for data compression in this chapter. The first three are simple encoding techniques, called: run-length, Huffman, and delta encoding. The last two are elaborate procedures that have established themselves as industry standards: LZW and JPEG.

Передача и хранение данных стоят денег. Подробная информация, с которой имеют дело, стоит дороже. Несмотря на это, наиболее цифровые данные не сохранены в наиболее компактной форме. Скорее, они сохранены любым способом, что делает их самыми простыми для использования, типа: текст ASCII от текстовых процессоров, двоичный код, который может быть выполнен на компьютере, индивидуальные выборки от системы сбора данных, и т.д. Как правило, эти удобные в работе методы кодирования требуют файлов данных вдвое большего размера, чем фактически необходимо, чтобы представить информацию. Сжатие Данных - общий термин для различных алгоритмов и программирует разработанный, чтобы адресовать эту проблему. Компрессионная программа используется, чтобы преобразовать данные от удобного в работе формата до одного оптимизированного для компактности. Аналогично, не-компрессионная программа возвращает информацию к ее первоначальной форме. Мы исследуем пять методов на сжатие данных в этой главе. Первые три - простые методы кодирования, называемые: run-length (выполняемая – длина; кодирование с переменной длиной строки), Huffman (Хаффман), и delta encoding (дельта кодирование). Последние два метода кодирования - сложные процедуры, которые утвердились как промышленные стандарты: LZW и JPEG.

Data Compression Strategies
Стратегии Сжатия Данных

Table 27-1 shows two different ways that data compression algorithms can be categorized. In (a), the methods have been classified as either **lossless** or **lossy**. A lossless technique means that the restored data file is *identical* to the original. This is absolutely necessary for many types of data, for example: executable code, word processing files, tabulated numbers, etc. You cannot afford to misplace even a single bit of this type of information. In comparison, data files that represent images and other acquired signals do not have to be keep in perfect condition for storage or transmission. All real world measurements inherently contain a certain amount of *noise*. If the changes made to these signals resemble a small amount of additional noise, no harm is done. Compression techniques that allow this type of degradation are called **lossy**. This distinction is

important because lossy techniques are much more effective at compression than lossless methods. The higher the compression ratio, the more noise added to the data.

В таблице 27-1 показано два различных способа, которыми алгоритмы компрессии данных могут быть категоризированы. В (a), методы были классифицированы как **lossless** (**без потерь**) или **lossy** (**с потерями**?). Методика без потерь означает, что восстановленный файл данных *идентичен* оригиналу. Это абсолютно необходимо для многих типов данных, например: выполнимый код, файлы обработки текстов, сведенные в таблицу числа, и т.д. Вы не можете позволять себе положить не на место даже единственный двоичный разряд(бит) этого типа информации. Для сравнения, файлы данных, которые представляют, изображения и другие приобретенные сигналы не должны быть, сохранены в отличном состоянии дляхранения или передачи. Все реальные(вещественные) мировые измерения неотъемлемо содержат некоторое количество шума. Если изменения, сделанные к этим сигналам походят на маленькое количество дополнительного шума, никакой вред не сделан. Компрессионные методы, которые позволяют этот тип деградации, называются lossy(поглощение). Это различие важно, потому что lossy методы намного более эффективны при сжатии чем методы без потерь. Чем выше компрессионное отношение(коэффициент), тем большее количество шума добавлялось к данным.

Lossless	Lossy
run-length	CS&Q
Huffman	JPEG
delta	MPEG
LZW	

a. Lossless or Lossy

Method	Group size: input	output
CS&Q	fixed	fixed
Huffman	fixed	variable
Arithmetic	variable	variable
run-length, LZW	variable	fixed

b. Fixed or variable group size

TABLE 27-1
Compression classifications. Data compression methods can be divided in two ways. In (a), the techniques are classified as *lossless* or *lossy*. Lossless methods restore the compressed data to exactly the same form as the original, while lossy methods only generate an approximation. In (b), the methods are classified according to a *fixed* or *variable* size of group taken from the original file and written to the compressed file.
ТАБЛИЦА 27-1
Компрессионные классификации. Методы сжатия данных могут быть разделены двумя способами. В (a), методы классифицированы как *lossless (*без потерь) или *lossy*(с потерями). Методы Без потерь восстанавливают сжатые данные к точно той же самой форме как оригинал, в то время как lossy методы генерируют только аппроксимацию. В (b), методы классифицированы согласно *фиксированному* или *переменному размеру группы*, принятой от первоначального файла и записанного в сжатый файл.

Images transmitted over the world wide web are an excellent example of why data compression is important. Suppose we need to download a digitized color photograph over a computer's 33.6 kbps modem. If the image is not compressed (a *TIFF* file, for example), it will contain about 600 kbytes of data. If it has been compressed using a *lossless* technique (such as used in the *GIF* format), it will be about one-half this size, or 300 kbytes. If *lossy* compression has been used (a JPEG file), it will be about 50 kbytes. The point is, the download times for these three equivalent files are 142 seconds, 71 seconds, and 12 seconds, respectively. That's a big difference! JPEG is the best choice for digitized photographs, while GIF is used with *drawn* images, such as company logos that have large areas of a single color.

Изображения, переданные по “Всемирной Паутине” - превосходный пример того, почему сжатие данных является важным. Предположим, что мы должны загрузить цифровую цветную фотографию в компьютер модем(модулятор-демодулятор) 33.6 kbps. Если изображение не сжато (например, *TIFF* файл), это будет содержать приблизительно 600 ки-

лобайтов данных. Если это было сжато, используя *методику без потерь(lossless)* (типа используемого в формате *GIF*), это будет около половины этого размера, или 300 килобайтов. Если использовалось сжатие *lossy(с потерями)* (JPEG файл), это будет приблизительно 50 килобайтов. Пункт, время загрузки для этих трех эквивалентных файлов - 142 секунды, 71 секунда, и 12 секунд, соответственно. Это - большая разность! JPEG - лучший выбор для цифровых фотографий, в то время как GIF (ФОРМАТ ОБМЕНА ГРАФИЧЕСКИМИ ДАННЫМИ) используется с выведенными(некоторыми) изображениями, типа эмблем компании, которые имеют большие области единственного(отдельного) цвета.

Our second way of classifying data compression methods is shown in Table 27-1b. Most data compression programs operate by taking a group of data from the original file, compressing it in some way, and then writing the compressed group to the output file. For instance, one of the techniques in this table is **CS&Q**, short for **coarser sampling and/or quantization**. Suppose we are compressing a digitized waveform, such as an audio signal that has been digitized to 12 bits. We might read two adjacent samples from the original file (24 bits), discard one of the sample completely, discard the least significant 4 bits from the other sample, and then write the remaining 8 bits to the output file. With 24 bits in and 8 bits out, we have implemented a 3:1 compression ratio using a lossy algorithm. While this is rather crude in itself, it is very effective when used with a technique called *transform compression*. As we will discuss later, this is the basis of JPEG.

Наш второй путь классификации методов сжатия данных показывается в таблице 27-1b. Большинство программ сжатия данных оперирует, беря группу данных от первоначального файла, сжимая это некоторым способом, и затем записывая сжатую группу в выходной файл. Например, один из методов в этой таблице **CS&Q**, короток **для более крупного осуществления выборки и/или квантования**. Предположим, что мы сжимаем цифровую форму волны, типа аудио-сигнала, который был оцифрован к 12 битам. Мы могли бы читать две смежных выборки из первоначального файла (24 бита), отказаться от одной из выборок полностью, отказаться от самых младших 4 битов другой выборки, и затем записать сохранение 8 битов в выходной файл. С 24 битами в и 8 битов из, мы осуществили коэффициент компрессии 3:1, используя алгоритм *с потерями*. В то время как это довольно грубо само по себе, это очень эффективно, когда используется с методикой называемой *сжатием трансформанты*. Как мы обсудим позже, это - основание JPEG.

Table 27-1b shows CS&Q to be a fixed-input fixed-output scheme. That is, a fixed number of bits are read from the input file and a smaller fixed number of bits are written to the output file. Other compression methods allow a variable number of bits to be read or written. As you go through the description of each of these compression methods, refer back to this table to understand how it fits into this classification scheme. Why are JPEG and MPEG not listed in this table? These are composite algorithms that combine many of the other techniques. They are too sophisticated to be classified into these simple categories.

Таблица 27-1b показывает CS&Q, чтобы быть схемой фиксированный ввод – фиксированный вывод. То есть установленное число битов читается от входного файла, и меньшее установленное число битов записывается в выходной файл. Другие компрессионные методы позволяют переменному числу битов читаться или записываться. Поскольку Вы проходите описание каждого из этих компрессионных методов, обратитесь назад к этой таблице, чтобы понять, как это вписываться в эту схему классификации. Почему - JPEG и MPEG, не перечислены в этой таблице? Они - составные алгоритмы, которые объединяют многие из других методов. Они слишком сложны, чтобы быть классифицированными в эти простые категории.

Run-Length Encoding
Кодирование с переменной длиной строки

Data files frequently contain the same character repeated many times in a row. For example, text files use multiple spaces to separate sentences, indent paragraphs, format tables & charts, etc. Digitized *signals* can also have runs of the same value, indicating that the signal is not changing. For instance, an image of the nighttime sky would contain long runs of the character or characters representing the black background. Likewise, digitized music might have a long run of zeros between songs. Run-length encoding is a simple method of compressing these types of files.

Файлы данных часто содержат один и тот же символ, повторяющийся в строке много раз. Например, текстовые файлы используют многочисленные пробелы, чтобы отделить предложения, параграфы отступа, таблицы формата и диаграммы, и т.д. Цифровые *сигналы* могут также иметь, выполняет того же самого значения, указывая, что сигнал не изменяется. Например, изображение ночного неба содержало бы, долго выполняется символа или символов, представляющих черный фон. Аналогично, цифровая музыка могла бы иметь длинным выполненным из нулей между песнями. Кодирование с переменной длиной строки - простой метод сжатия этих типов файлов.

Figure 27-1 illustrates run-length encoding for a data sequence having frequent runs of *zeros*. Each time a zero is encountered in the input data, *two* values are written to the output file. The first of these values is a zero, a flag to indicate that run-length compression is beginning. The second value is the number of zeros in the run. If the average run-length is longer than two, compression will take place. On the other hand, many single zeros in the data can make the encoded file larger than the original.

Рисунок 27-1 иллюстрирует кодирование с переменной длиной строки для последовательности данных, имеющих частое выполнение нулей. Каждый раз с нулем сталкиваются во входных данных, два значения написаны к выходному файлу. Первый из этих значений - нуль, флажок, чтобы указать, что сжатие выполняемой - длины начинается. Второе значение - число нулей в выполненном. Если средняя выполняемая - длина дольше чем два, сжатие будет иметь место. С другой стороны, много единственных(отдельных) нулей в данных могут делать закодированный файл большим чем оригинал.

Many different run-length schemes have been developed. For example, the input data can be treated as individual bytes, or groups of bytes that represent something more elaborate, such as floating point numbers. Run-length encoding can be used on only *one* of the characters (as with the *zero* above), *several* of the characters, or *all* of the characters.

Много различных схем выполняемой - длины были разработаны. Например, входные данные могут быть обработаны как индивидуальные байты, или группы байтов, которые представляют кое-что более сложное, типа чисел с плавающей запятой. Кодирование с переменной длиной строки может использоваться на только одном из символов (как в случае с *нулем*, описанном выше), *нескольких* из символов, или *всех* символов.

A good example of a generalized run-length scheme is **PackBits**, created for Macintosh users. Each byte (eight bits) from the input file is replaced by nine bits in the compressed file. The added ninth bit is interpreted as the *sign* of the number. That is, each character read from the in-

put file is between 0 to 255, while each character written to the encoded file is between -255 and 255. To understand how this is used, consider the input file: 1, 2, 3, 4, 2, 2, 2, 2, 4 and the compressed file generated by the PackBits algorithm: The 1, 2, 3, 4, 2, -3, 4. The compression program simply transfers each number from the input file to the compressed file, with the exception of the run: 2,2,2,2. This is represented in the compressed file by the two numbers: 2,-3. The first number ("2") indicates what character the run consists of. The second number ("-3") indicates the number of characters in the run, found by taking the absolute value and adding one. For instance, 4, -2 means 4,4,4; 21,-4 means 21,21,21,21,21, etc.

Хороший пример обобщенной схемы выполняемой - длины - **PackBits**, созданный для пользователей Macintosh. Каждый байт (восемь битов) от входного файла заменен девятью битами в сжатом файле. Добавленный девятый двоичный разряд(бит) интерпретируется как *знак* номера. То есть каждое символьное чтение от входного файла - между от 0 до 255, в то время как каждый символ, записанный в закодированный файл - между -255 и 255. Чтобы понимать, как это используется, рассмотрите входной файл: 1, 2, 3, 4, 2, 2, 2, 2, 4. Сжатый файл, сгенерированный PackBits алгоритмом: 1, 2, 3, 4, 2, - 3, 4. Компрессионная программа просто передает(перемещает) каждый номер от входного файла до сжатого файла, за исключением выполненного: 2,2,2,2. Это представлено в сжатом файле этими двумя числами(номерами): 2, -3. Первый номер ("2") указывает то, из какого символ выполненный состоит. Второй номер ("-3") указывает число символов в выполненном, найденном, беря абсолютное значение и добавляя один. Например, 4, -2 средства 4,4,4; 21, - 4 21,21,21,21,21 средства, и т.д.

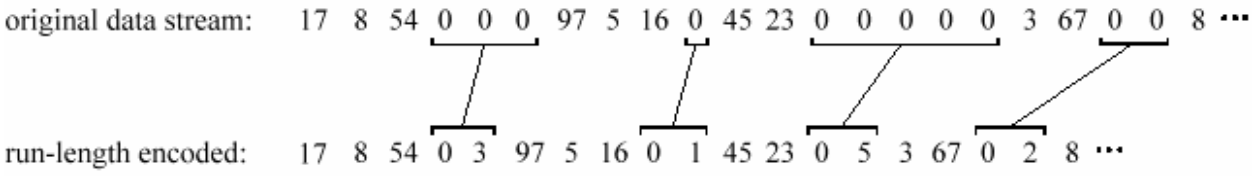


FIGURE 27-1
Example of run-length encoding. Each run of zeros is replaced by two characters in the compressed file: a zero to indicate that compression is occurring, followed by the number of zeros in the run.

An inconvenience with PackBits is that the nine bits must be reformatted into the standard eight bit bytes used in computer storage and transmission. A useful modification to this scheme can be made when the input is restricted to be ASCII text. As shown in Table 27-2, each ASCII character is usually stored as a full byte (eight bits), but really only uses *seven* of the bits to identify the character. In other words, the values 127 through 255 are not defined with any standardized meaning, and do not need to be stored or transmitted. This allows the eighth bit to indicate if run-length encoding is in progress.

Неудобство с PackBits - то, что эти девять битов должны быть переформатированы в стандартные байты с восемью двоичными разрядами, используемые в компьютерной памяти(хранении) и передаче. Полезная модификация к этой схеме может быть сделана, когда ввод ограничен, чтобы быть текстом ASCII. Как показано в таблице 27-2, каждый символ ASCII обычно сохраняется как полный байт (восемь битов), но действительно только использует семь из битов, чтобы идентифицировать символ. Другими словами, значения 127 до 255 не определены любым стандартизированным значением, и не должны быть сохранены или переданы. Это позволяет восьми разрядам(биту) указывать, происходит ли кодирование с переменной длиной строки.

0	Null	32	space	64	@	96	`
1	Start heading	33	!	65	A	97	A
2	Start of text	34	"	66	B	98	B
3	end of text	35	#	67	C	99	C
4	end of xmit	36	$	68	D	100	D
5	enquiry	37	%	69	E	101	E
6	acknowledge	38	&	70	F	102	F
7	bell, beep	39	'	71	G	103	G
8	backspace	40	(	72	H	104	H
9	Horz. tab	41	)	73	I	105	I
10	line feed	42	*	74	J	106	J
11	vert. tab, home	43	+	75	K	107	K
12	Form feed, cls	44	,	76	L	108	L
13	carriage return	45	-	77	M	109	M
14	Shift out	46	.	78	N	110	N
15	Shift in	47	/	79	O	111	O
16	data line esc	48	0	80	P	112	P
17	device control 1	49	1	81	Q	113	Q
18	device control 2	50	2	82	R	114	R
19	device control 3	51	3	83	S	115	S
20	device control 4	52	4	84	T	116	t
21	negative ack.	53	5	85	U	117	r
22	synch. idle	54	6	86	V	118	v
23	end xmit block	55	7	87	W	119	w
24	cancel	56	8	88	X	120	x
25	end of medium	57	9	89	Y	121	y
26	substitute	58	:	90	Z	122	z
27	escape	59	;	91	[	123	{
28	file separator	60	<	92	\	124	|
29	group separator	61	=	93	]	125	}
30	record separator	62	>	94	^	126	~
31	unit separator	63	?	95	_	127	del

TABLE 27-2
ASCII codes. This is a long established standard for allowing letters and numbers to be represented in digital form. Each printable character is assigned a number between 32 and 127, while the numbers between 0 and 31 are used for various control actions. Even though only 128 codes are defined, ASCII characters are usually stored as a full byte (8 bits). The undefined values (128 to 255) are often used for Greek letters, math symbols, and various geometric patterns; however, this is not standardized. Many of the control characters (0 to 31) are based on older communications networks, and are not applicable to computer technology.

ТАБЛИЦА 27-2
Коды ASCII. Это - длинный установленный стандарт для разрешения символов и чисел, которые будут представлены в цифровой форме. Каждому печатаемому символу назначен номер между 32 и 127, в то время как номера между 0 и 31 используются для различных управляющих воздействий. Даже при том, что только 128 кодов определены, символы ASCII обычно сохраняются как полный байт (8 битов). Неопределенные значения (от 128 до 255) часто используются для греческих символов, математических символов, и различных геометрических образцов; однако, это не стандартизировано. Многие из управляющих символов (от 0 до 31) основаны на старших системах коммуникаций(сетях), и не применимы к компьютерной технологии.

Huffman Encoding
Кодирование Хаффмана

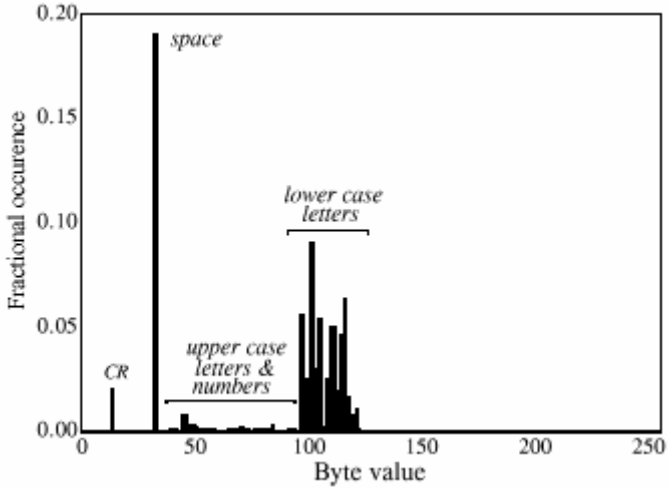
This method is named after D.A. Huffman, who developed the procedure in the 1950s. Figure 27-2 shows a histogram of the byte values from a large ASCII file. More than 96% of this file consists of only 31 characters: the lower case letters, the space, the comma, the period, and the carriage return. This observation can be used to make an appropriate compression scheme for this file. To start, we will assign each of these 31 common characters a five bit binary code:

00000 = "a", 00001 = "b", 00010 = "c", etc. This allows 96% of the file to be reduced in size by 5/8. The last of the five bit codes, 11111, will be a flag indicating that the character being transmitted is not one of the 31 common characters. The next eight bits in the file indicate what the character is, according to the standard ASCII assignment. This results in 4% of the characters in the input file requiring 5+8=13 bits. The idea is to assign frequently used characters fewer bits, and seldom used characters more bits. In this example, the *average* number of bits required per original character is: 0.96 x 5 + 0.04 x 13 = 5.32. In other words, an overall compression ratio of: 8 bits/5.32 bits, or about . 1.5 : 1

Этот метод назван по имени D.A. Huffman, кто разработал процедуру в 1950-ых. Рисунок 27-2 показывает гистограмму значений байта от большого файла ASCII. Более чем 96% этого файла состоят только из 31 символа: символы строчных букв, пробел, запятая, точка, и возврат каретки. Это наблюдение может использоваться, чтобы делать соответствующую компрессионную схему этого файла. Чтобы начать, мы назначим каждому из этих 31 общих символов двоичный код пять двоичных разрядов: 00000 = "a", 00001 = "b", 00010 = "c", и т.д. Это позволяет 96 % файла быть сокращенным в размере 5/8. Последний из кодов пять двоичных разрядов, 11111, будет флажок, указывающий, что передаваемый символ - ни один из 31 общих символов. Следующие восемь битов в файле указывают, каков символ, согласно стандартному назначению ASCII. Это приводит к 4 % символов во входном файле, требующем 5+8=13 битов. Идея состоит в том, чтобы назначить часто используемым символам меньшее количество битов, и редко используемым символам большее количество битов. В этом примере, *среднее* число битов, требуемых на первоначальный символ: 0.96 x 5 + 0.04 x 13 = 5.32. Другими словами, полный компрессионный коэффициент: 8 битов/5.32 бита, или в отношении 1.5: 1.

FIGURE 27-2
Histogram of text. This is a histogram of the ASCII values from a chapter in this book. The most common characters are the lower case letters, the space and the carriage return.

РИСУНОК 27-2
Гистограмма текста. Это - гистограмма значений ASCII от главы в этой книге. Наиболее обычные символы - символы строчных букв, пробел и возврат каретки.



Huffman encoding takes this idea to the extreme. Characters that occur most often, such the space and period, may be assigned as few as one or two bits. Infrequently used characters, such as: !, @, #, $ and ?, may require a dozen or more bits. In mathematical terms, the optimal situation is reached when the number of bits used for each character is proportional to the logarithm of the character's probability of occurrence.

Кодирование Хаффмана берет эту идею до крайности. Символам, которые встречаются наиболее часто, таким как пробел и точка, могут быть назначены только один или два бита. Нечасто используемые символы, типа: !, @, #, $ и ?, могут требовать дюжины или большего количества битов. В математических терминах, оптимальное положение(ситуация) достигнуто, когда число битов, используемых для каждого символа пропорционально к логарифму вероятности местонахождения символа.

A clever feature of Huffman encoding is how the variable length codes can be packed together. Imagine receiving a serial data stream of ones and zeros. If each character is represented by eight bits, you can directly separate one character from the next by breaking off 8 bit chunks. Now consider a Huffman encoded data stream, where each character can have a variable number of bits. How do you separate one character from the next? The answer lies in the proper selection of the Huffman codes that enable the correct separation. An example will illustrate how this works.

Умная(талантливая; искусная) особенность кодирования Хаффмана - то, как коды переменной длины могут быть упакованы вместе. Вообразите получение последовательного потока данных единиц и нулей. Если каждый символ представлен восьмью битами, Вы можете непосредственно отделить один символ от следующего, прерывая куски 8 двоичными разрядами. Теперь рассмотрите кодирование Хаффмана потока данных, где каждый символ может иметь переменное число битов. Как Вы отделите один символ от следующего? Ответ лежит в надлежащем выборе кодирования Хаффмана, которое позволяет правильное разделение. Пример иллюстрирует, как это работает.

Figure 27-3 shows a simplified Huffman encoding scheme. The characters *A* through *G* occur in the original data stream with the probabilities shown. Since the character *A* is the most common, we will represent it with a single bit, the code: 1. The next most common character, *B*, receives two bits, the code: 01. This continues to the least frequent character, *G*, being assigned six bits, 000011. As shown in this illustration, the variable length codes are resorted into eight bit groups, the standard for computer use.

Рисунок 27-3 показывает упрощенную схему кодирования Хаффмана. Символы от A до G встречаются в первоначальном потоке данных с показанными вероятностями. Так как символ A наиболее обычен, мы представим его единственным двоичным разрядом, код: 1. Следующий наиболее обычный символ, B, получает два бита, код: 01. Это продолжается к наименее частому символу, G, назначаемому шесть битов, 000011. Как показано в этой иллюстрации, коды переменной длины обращаются в группы по восемь двоичных разрядов, стандарт для компьютерного использования.

FIGURE 27-3
Huffman encoding. The encoding table assigns each of the seven letters used in this example a variable length binary code, based on its probability of occurrence. The original data stream composed of these 7 characters is translated by this table into the Huffman encoded data. Since each of the Huffman codes is a different length, the binary data need to be regrouped into standard 8 bit bytes for storage and transmission.

Example Encoding Table

letter	probability	Huffman code
A	.154	1
B	.110	01
C	.072	0010
D	.063	0011
E	.059	0001
F	.015	000010
G	.011	000011

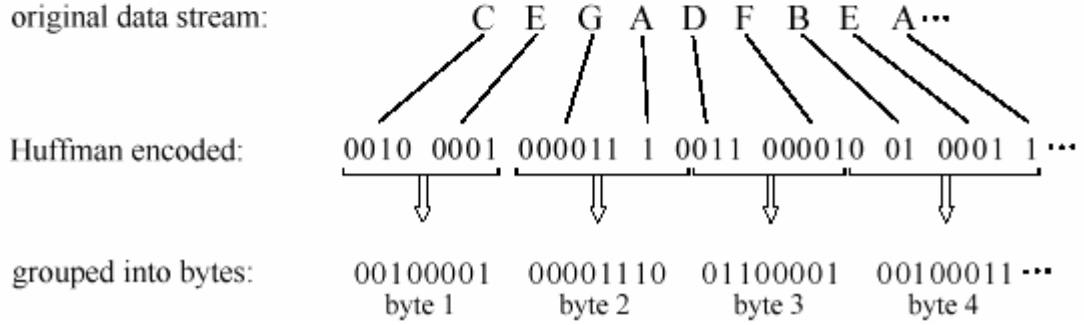


FIGURE 27-3

Huffman encoding. The encoding table assigns each of the seven letters used in this example a variable length binary code, based on its probability of occurrence. The original data stream composed of these 7 characters is translated by this table into the Huffman encoded data. Since each of the Huffman codes is a different length, the binary data need to be regrouped into standard 8 bit bytes for storage and transmission.
РИСУНОК 27-3
Кодирование Хаффмана. Таблица кодирования назначает каждому из семи символов, используемых в этом примере двоичный код переменной длины, основанный на его вероятности местонахождения. Первоначальный поток данных, составленный из этих 7 символов, оттранслирован этой таблицей в Хаффмана закодированные данные. Так как каждый из кодов Хаффмана - различная длина, двоичные данные должны быть перегруппированы в стандарт 8 двоичных разрядов на байт, для хранения и передачи.

When uncompression occurs, all the eight bit groups are placed end-to-end to form a long serial string of ones and zeros. Look closely at the encoding table of Fig. 27-3, and notice how each code consists of two parts: a number of zeros before a *one*, and an optional binary code after the *one*. This allows the binary data stream to be separated into codes without the need for delimiters or other marker between the codes. The uncompression program looks at the stream of ones and zeros until a valid code is formed, and then starting over looking for the next character. The way that the codes are formed insures that no ambiguity exists in the separation.

Когда не-компрессия происходит, все группы по восемь двоичных разрядов помещены от начала до конца, формируя длинную последовательную строку из единиц и нулей. Рассмотрите ближе таблицу кодирования на рис. 27-3, и обратите внимание, как каждый код состоит из двух частей: ряд нулей перед *единицей*, и необязательный двоичный код после единицы. Это позволяет двоичному потоку данных быть отделенным в коды без потребности в разделителях или другом маркере между кодами. Некомпрессионная программа следит за потоком единиц и нулей пока не будет сформирован справедливый код, и затем начинает поиск следующего символа. Путь, которым коды сформированы, обеспечивает, чтобы в разделении не существовало никакой неоднозначности.

A more sophisticated version of the Huffman approach is called **arithmetic encoding**. In this scheme, *sequences* of characters are represented by individual codes, according to their probability of occurrence. This has the advantage of better data compression, say 5-10%. Run-length encoding followed by either Huffman or arithmetic encoding is also a common strategy. As you might expect, these types of algorithms are very complicated, and usually left to data compression specialists.

Более сложная версия подхода Хаффмана называется **арифметическим кодированием**. В этой схеме, *последовательности* символов представлены индивидуальными кодами, согласно их вероятности местонахождения. Это имеет преимущество лучшего сжатия данных, скажем 5-10%. Кодирование длин серий сопровождаемое или Хаффманом или арифметическим кодированием - также обычная стратегия. Поскольку Вы могли бы ожидать, эти типы алгоритмов очень усложнены, и обычно оставлены специалистам сжатия данных.

To implement Huffman or arithmetic encoding, the compression and un-compression algorithms must agree on the binary codes used to represent each character (or groups of characters). This can be handled in one of two ways. The simplest is to use a predefined encoding table that is always the same, regardless of the information being compressed. More complex schemes use encoding optimized for the particular data being used. This requires that the encoding table be included in the compressed file for use by the uncompression program. Both methods are common.

Чтобы осуществлять кодирование Хаффмана или арифметическое кодирование, алгоритмы сжатия и не-сжатия должны договориться о двоичных кодах, обычно представляющих

каждый символ (или группы символов). Это может быть обработано способом из двух путей. Самый простой, чтобы использовать предопределенную таблицу кодирования, которая является всегда тем же самый, независимо от сжимаемой информации. Более комплексное использование схем кодирования оптимизировано для специфических используемых данных. Это требует, чтобы таблица кодирования была включена в сжатый файл для использования не-компрессионной программой. Оба метода общие(обычны).

Delta Encoding
Дельта Кодирование

In science, engineering, and mathematics, the Greek letter *delta* (**)) is used to denote the *change* in a variable. The term *delta encoding*, refers to several techniques that store data as the *difference* between successive samples (or characters), rather than directly storing the samples themselves. Figure 27-4 shows an example of how this is done. The first value in the delta encoded file is the same as the first value in the original data. All the following values in the encoded file are equal to the difference (delta) between the corresponding value in the input file, and the *previous* value in the input file.

В науке, конструировании, и математике, символ греческая *дельта* (Δ) используется, чтобы обозначить изменение в переменной. Термин *дельта кодирование*, относится к нескольким методам, которые сохраняют данные как разность между последовательными выборками (или символы), скорее, чем непосредственно сохранение выборок непосредственно. Рисунок 27-4 показывает пример того, как это сделано. Первое значение в файле дельта кодирования - то же самое как первое значение в первоначальных данных. Все следующие значения в закодированном файле равны разности (дельта) между соответствующим значением во входном файле, и *предыдущим* значением во входном файле.

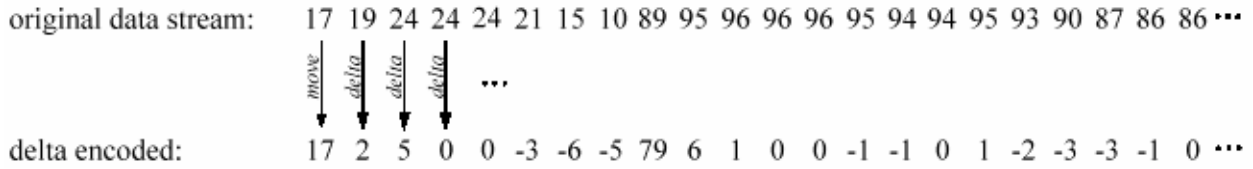


FIGURE 27-4
Example of delta encoding. The first value in the encoded file is the same as the first value in the original file. Thereafter, each sample in the encoded file is the difference between the current and last sample in the original file.
РИСУНОК 27-4
Пример дельта кодирования. Первое значение в закодированном файле - то же самое как первое значение в первоначальном файле. После этого, каждая следующая выборка в закодированном файле - разность между текущей и последней выборкой в первоначальном файле.

Delta encoding can be used for data compression when the values in the original data are *smooth*, that is, there is typically only a small change between adjacent values. This is not the case for ASCII text and executable code; however, it is very common when the file represents a *signal*. For instance, Fig. 27-5a shows a segment of an audio signal, digitized to 8 bits, with each sample between -127 and 127. Figure 27-5b shows the delta encoded version of this signal. The key feature is that the delta encoded signal has a *lower amplitude* than the original signal. In other words, delta encoding has increased the probability that each sample's value will be near zero, and decreased the probability that it will be far from zero. This uneven probability is just the thing that Huffman encoding needs to operate. If the original signal is not changing, or is changing in a straight line, delta encoding will result in runs of samples having the same value.

Дельты кодирование может использоваться для сжатия данных, когда значения в первоначальных данных *гладки*, то есть имеется типично только маленькие изменения между смежными значениями. Не так обстоит дело для текста ASCII и выполнимого кода; однако, очень обычно, когда файл представляет сигнал. Например, рис. 27-5a показывает сегмент аудио-сигнала, оцифрованного 8 бит на каждую выборку между -127 и 127. Рисунок 27-5b показывает дельта кодированную версию этого сигнала. Ключевая особенность - то, что дельта кодированный сигнал имеет более низкую амплитуду, чем первоначальный сигнал. Другими словами, дельта кодирование увеличило вероятность, что значение каждой выборки будет около нуля, и уменьшило вероятность, что это будет далеко(дальше) от нуля. Эта нечетная вероятность - только случай(обстоятельство) что требуется операция кодированием Хаффмана. Если первоначальный сигнал не изменяется, или изменяется в прямой строке(линии), дельта кодирование будет выполнением выборок, имеющих то же самое значение(изменения значения прямой – закодированным графиком).

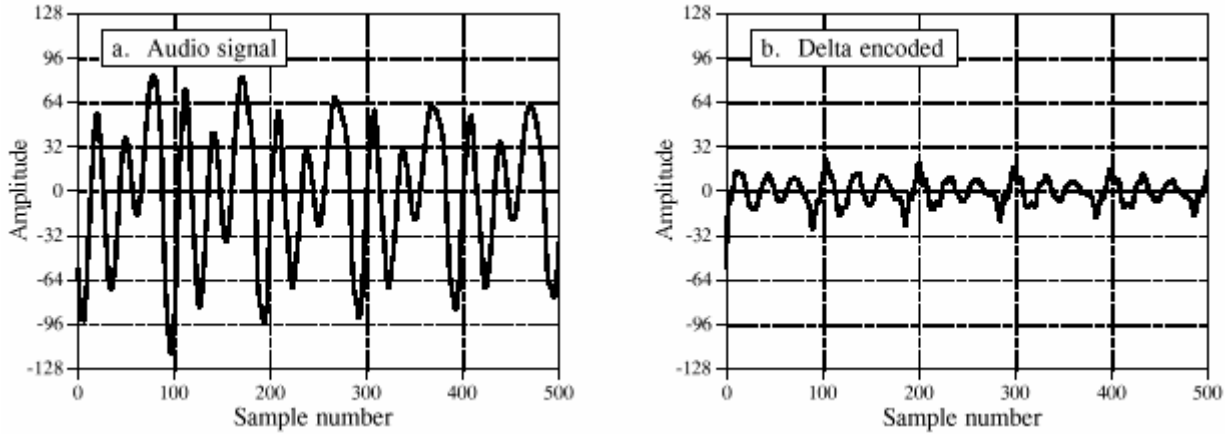


FIGURE 27-5
Example of delta encoding. Figure (a) is an audio signal digitized to 8 bits. Figure (b) shows the delta encoded version of this signal. Delta encoding is useful for data compression if the signal being encoded varies slowly from sample-to-sample.

РИСУНОК 27-5
Пример дельта кодирования. Рисунок (a) – аудио-сигнал, оцифрованный к 8 битам. Рисунок (b) показывает дельта кодированную версию этого сигнала. Дельта кодирование полезно для сжатия данных, если закодированный сигнал изменяется медленно от " выборки к выборке ".

This is what run-length encoding requires. Correspondingly, delta encoding followed by Huffman and/or run-length encoding is a common strategy for compressing signals.

Это - то, чего кодирование с переменной длиной строки требует. Соответственно, дельта, кодирование сопровождаемое Человеком и кодированием длин серий с переменной длиной строки - обычная стратегия для сжатия сигналов.

The idea used in delta encoding can be expanded into a more complicated technique called **Linear Predictive Coding**, or **LPC**. To understand LPC, imagine that the first 99 samples from the input signal have been encoded, and we are about to work on sample number 100. We then ask ourselves: based on the first 99 samples, what is the most likely value for sample 100? In delta encoding, the answer is that the most likely value for sample 100 is the same as the previous value, sample 99. This expected value is used as a reference to encode sample 100. That is, the *difference* between the sample and the expectation is placed in the encoded file. LPC expands on this by making a better guess at what the most probable value is. This is done by looking at the last several samples, rather than just the last sample. The algorithms used by LPC are similar to recursive filters, making use of the z-transform and other intensively mathematical techniques.

Идея, используемая в дельта кодировании может быть расширена в более сложную методику по имени **Линейное Прогнозирующее(Предиктивное) Кодирование**, или (**ЛПК**). Чтобы понимать ЛПК, вообразите, что первые 99 выборок от входного сигнала были закодированы, и мы собираемся работать на выборке номер 100. Мы тогда спрашиваем себя: базируясь на первых 99 выборках, что является наиболее вероятным значением для выборки 100? В дельта кодировании, ответ - то, что наиболее вероятное значение для выборки 100 является тем же самый как предыдущее значение, выборка 99. Это ожидаемое значение используется как ссылка, чтобы кодировать выборку 100. То есть, эта *разность* между выборкой и ожидаемой выборкой помещена в закодированный файл. ЛПК подробно останавливается, это, делая лучше приблизительно определяет, каково наиболее вероятное значение. Это сделано смотрящий на последние(прошлые) несколько выборок, скорее чем только последняя(прошлая) выборка. Алгоритмы, используемые ЛПК подобны рекурсивным фильтрам, используя z-трансформанту и другие интенсивно математические методы.

LZW Compression
LZW Сжатие

LZW compression is named after its developers, A. Lempel and J. Ziv, with later modifications by Terry A. Welch. It is the foremost technique for general purpose data compression due to its simplicity and versatility. Typically, you can expect LZW to compress text, executable code, and similar data files to about one-half their original size. LZW also performs well when presented with extremely redundant data files, such as tabulated numbers, computer source code, and acquired signals. Compression ratios of 5:1 are common for these cases. LZW is the basis of several personal computer utilities that claim to *"double the capacity of your hard drive."*

LZW сжатие названо по имени его разработчиков, A. Lempel и J. Ziv, с более поздними модификациями Terry A. Welch (Тэрри А. Велчом). Это - передовая методика для универсального сжатия данных из-за его простоты и многосторонности. Как правило, Вы можете ожидать, что LZW сожмет текст, выполнимый код, и подобные файлы данных примерно к половине их первоначального размера. LZW также исполняет хорошо когда представлено с чрезвычайно избыточными файлами данных, типа: сведенных в таблицу чисел, компьютерного исходного текста, и приобретенных сигналов. Компрессионные отноше-

ния(коэффициенты) 5:1 обычны для этих случаев. LZW - основание нескольких утилит персонального компьютера, которые утверждают, что "*удвоили вместимость вашего жесткого диска*".

LZW compression is always used in GIF image files, and offered as an option in TIFF and PostScript. LZW compression is protected under U.S. patent number 4,558,302, granted December 10, 1985 to Sperry Corporation (now the Unisys Corporation). For information on commercial licensing, contact: Welch Licensing Department, Law Department, M/SC2SW1, Unisys Corporation, Blue Bell, Pennsylvania, 19424-0001.

LZW сжатие всегда используется в загрузочных модулях GIF(.GIF файлах), и предлагает как опцию в TIFF и PostScript. LZW сжатие защищено Американским патентом номером 4558302, выданного 10 декабря, 1985 корпорации Sperry (теперь корпорация Unisys). Для информации относительно лицензирования рекламы, войдите в контакт: Welch Licensing Department, Law Department, M/SC2SW1, Unisys Corporation, Blue Bell, Pennsylvania, 19424-0001.

LZW compression uses a **code table**, as illustrated in Fig. 27-6. A common choice is to provide 4096 entries in the table. In this case, the LZW encoded data consists entirely of 12 bit codes, each referring to one of the entries in the code table. Uncompression is achieved by taking each code from the compressed file, and translating it through the code table to find what character or characters it represents. Codes 0-255 in the code table are always assigned to represent single bytes from the input file. For example, if only these first 256 codes were used, each byte in the original file would be converted into 12 bits in the LZW encoded file, resulting in a 50% larger file size. During uncompression, each 12 bit code would be translated via the code table back into the single bytes. Of course, this wouldn't be a useful situation.

LZW сжатие использует **таблицу кода**, как иллюстрировано в рис. 27-6. Обычный выбор должен обеспечить 4096 входов в таблице. В этом случае, LZW кодирование данных, состоит полностью из кодов 12 двоичных разрядов, каждый что касается(сформирован; относится) к одному из входов в таблице кода. Несжатие достигнуто, беря каждый код от сжатого файла, и транслируя это через таблицу кода, чтобы найти то, какой символ или символы это представляет. Кодирование 0-255 в таблице кода, всегда поручаются представить единственные(отдельные) байты от входного файла. Например, если бы только эти первые 256 кодов использовались, каждый байт в первоначальном файле был бы преобразован в 12 битный закодированный в LZW файл, приводя большой размер файла к 50%. В течение несжатия, каждый код 12 двоичных разрядов был бы оттранслирован через таблицу кода назад в единственные(отдельные) байты. Конечно, это не было бы полезное положение(ситуация).

FIGURE 27-6
Example of code table compression. This is the basis of the popular LZW compression method. Encoding occurs by identifying sequences of bytes in the original file that exist in the code table. The 12 bit code representing the sequence is placed in the compressed file instead of the sequence. The first 256 entries in the table correspond to the single byte values, 0 to 255, while the remaining entries correspond to *sequences* of bytes. The LZW algorithm is an efficient way of generating the code table based on the particular data being compressed. (The code table in this figure is a simplified example, not one actually generated by the LZW algorithm).

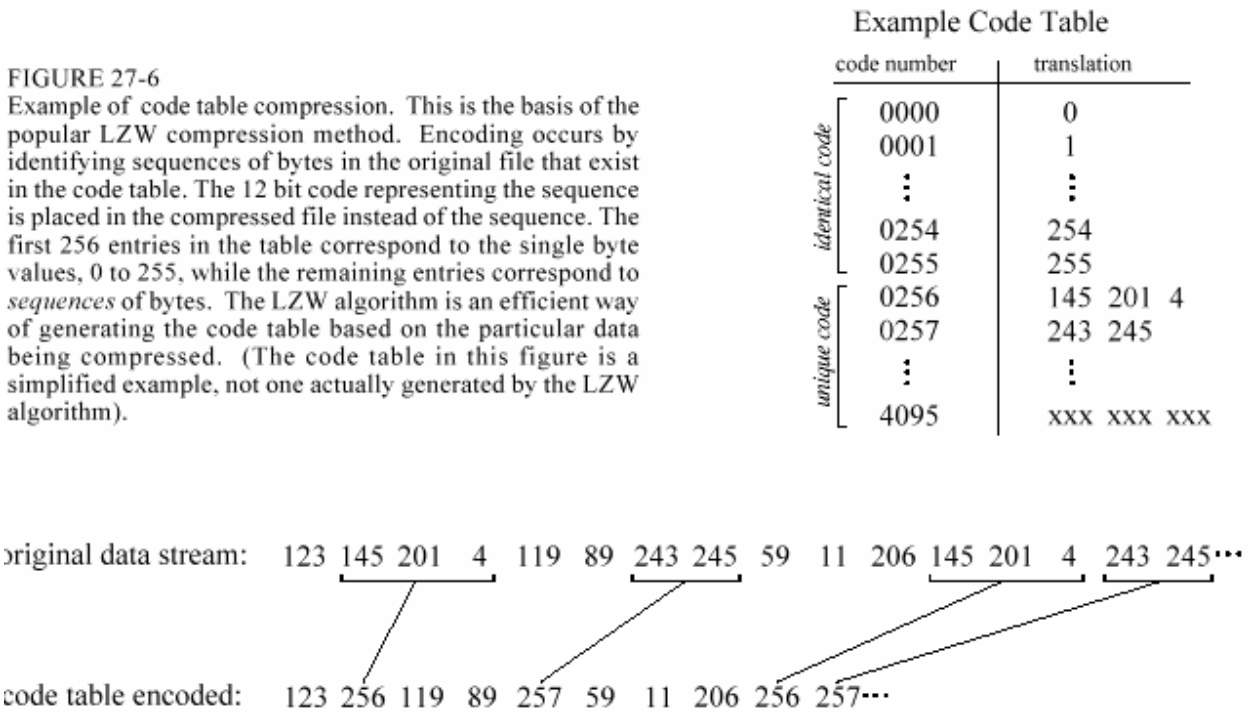


FIGURE 27-6
Example of code table compression. This is the basis of the popular LZW compression method. Encoding occurs by identifying sequences of bytes in the original file that exist in the code table. The 12 bit code representing the sequence is placed in the compressed file instead of the sequence. The first 256 entries in the table correspond to the single byte values, 0 to 255, while the remaining entries correspond to *sequences* of bytes. The LZW algorithm is an efficient way of generating the code table based on the particular data being compressed. (The code table in this figure is a simplified example, not one actually generated by the LZW algorithm).
РИСУНОК 27-6
Пример таблицы сжатия кода. Это - основание популярного компрессионного метода LZW. Кодирование происходит, идентифицируя последовательности байтов в первоначальном файле, которые существуют в таблице кода. Коды 12 двоичных разрядов, представляющих последовательность, помещаются в сжатый файл вместо последовательности. Первые 256 входов в таблице соответствуют единственным(отдельным) значениям байта, от 0 до 255, в то время как остающиеся входы соответствуют последовательностям байтов. LZW алгоритм - эффективный путь производства таблицы кода, основанной на специфических данных, будучи сжимаемых. (Таблица кода в этом рисунке - упрощенный пример, не один фактически сгенерированный LZW алгоритмом).

The LZW method achieves compression by using codes 256 through 4095 to represent *sequences* of bytes. For example, code 523 may represent the sequence of three bytes: 231 124 234. Each time the compression algorithm encounters this sequence in the input file, code 523 is placed in the encoded file. During uncompression, code 523 is translated via the code table to recreate the true 3 byte sequence. The longer the sequence assigned to a single code, and the more often the sequence is repeated, the higher the compression achieved.

LZW метод достигает сжатия, используя коды 256 до 4095, чтобы представить последовательности байтов. Например, код 523 может представлять последовательность трех байтов: 231 124 234. Каждый раз алгоритм компрессии сталкивается с этой последовательностью во входном файле, код 523 помещен в закодированный файл. В течение несжатия, код 523 оттранслирован через таблицу кода, чтобы освежить(пересоздать) истинную последовательность 3 байта. Чем длиннее последовательность, назначенная на отдельный код, и чем более часто последовательность повторена, тем выше достигнутое сжатие.

Although this is a simple approach, there are two major obstacles that need to be overcome: (1) how to determine what sequences should be in the code table, and (2) how to provide the un-

compression program the same code table used by the compression program. The LZW algorithm exquisitely solves both these problems.

Хотя это - простой подход, имеются два главных препятствия, которые должны быть преодолены: (1), как определять, что последовательности должны быть в таблице кода, и (2), как обеспечить программу не-сжатия, той же самой таблицей кода, используемой компрессионной программой. LZW алгоритм изящно решает обе эти проблемы.

When the LZW program starts to encode a file, the code table contains only the first 256 entries, with the remainder of the table being blank. This means that the first codes going into the compressed file are simply the single bytes from the input file being converted to 12 bits. As the encoding continues, the LZW algorithm identifies repeated sequences in the data, and adds them to the code table. Compression starts the second time a sequence is encountered. The key point is that a sequence from the input file is not added to the code table until it has already been placed in the compressed file as individual characters (codes 0 to 255). This is important because it allows the uncompression program to *reconstruct* the code table directly from the compressed data, without having to transmit the code table separately.

Когда программа LZW начинает кодировать файл, таблица кода содержит только первые 256 входов, с остаточным членом от таблицы, являющейся пробелом. Это означает, что первые коды, входящие в сжатый файл - просто единственные(отдельные) байты от входного файла, преобразовываемые к 12 битам. Поскольку кодирование продолжается, алгоритм LZW идентифицирует повторные последовательности в данных, и прибавляет их к таблице кода. Сжатие начинается, когда последовательность встречается во второй раз. Ключевой пункт - то, что последовательность от входного файла не добавлена к таблице кода, пока это уже не было помещено в сжатый файл как индивидуальные символы (коды от 0 до 255). Это важно, потому что это позволяет некомпрессионной программе восстанавливать таблицу кода непосредственно от сжатых данных, без того, чтобы иметь необходимость передавать таблицу кода отдельно.

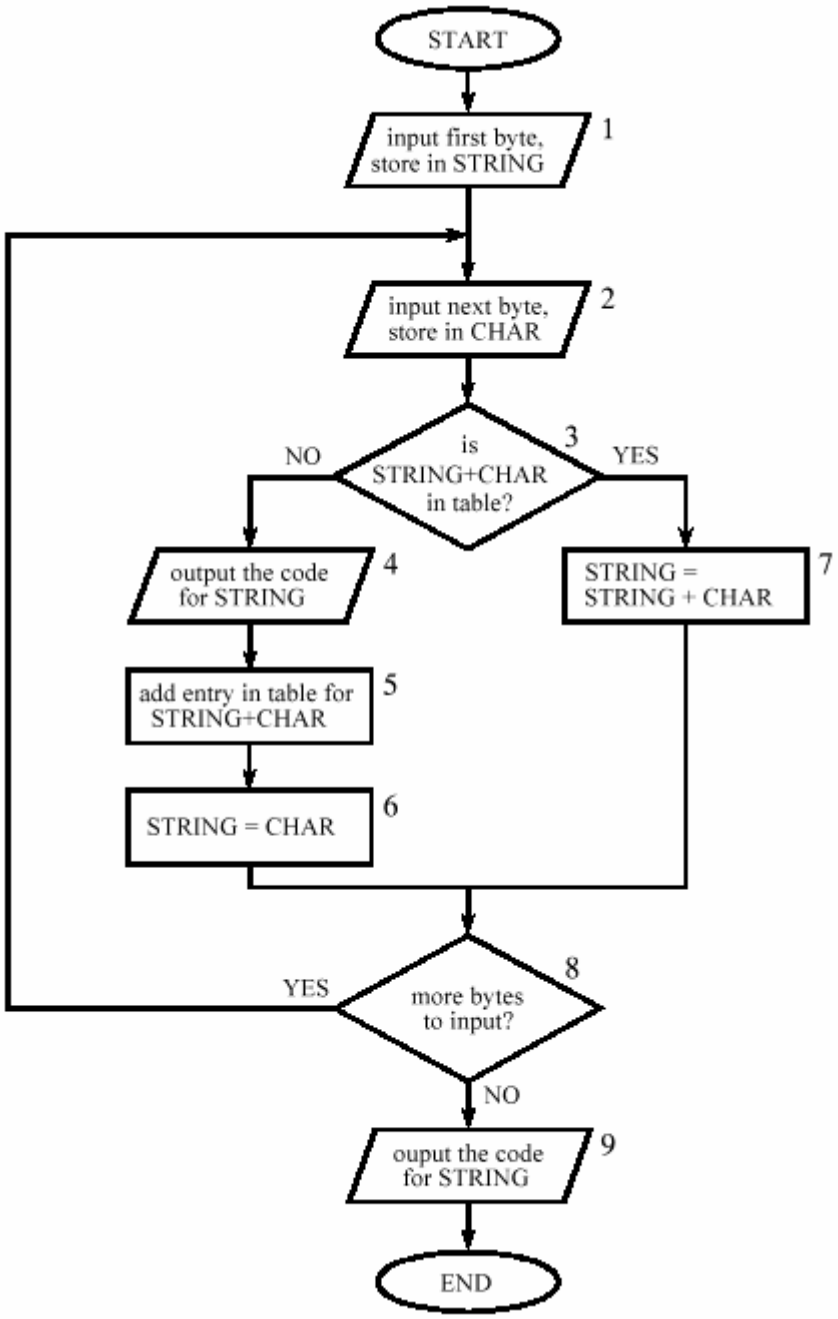


FIGURE 27-7
LZW compression flowchart. The variable, *CHAR*, is a single byte. The variable, *STRING*, is a variable length sequence of bytes. Data are read from the input file (box 1 & 2) as single bytes, and written to the compressed file (box 4) as 12 bit codes. Table 27-3 shows an example of this algorithm.
Рисунок 27-7
Блок-схема компрессии LZW. Переменная, *CHAR*, есть единственный(отдельный) байт. Переменная, *STRING*, является переменной длинной последовательности байтов. Данные читаются от входного файла(box 1 & 2) как одиночные байты, и записываются в соответствующий сжатый файл (box 4) как 12 разрядные коды.

Figure 27-7 shows a flowchart for LZW compression. Table 27-3 provides the step-by-step details for an example input file consisting of 45 bytes, the ASCII text string: *the/rain/in/Spain/falls/mainly/on/the/plain*. When we say that the LZW algorithm reads the character "a" from the input file, we mean it reads the value: 01100001 (97 expressed in 8 bits), where 97 is "a" in ASCII. When we say it writes the character "a" to the encoded file, we mean it writes: 000001100001 (97 expressed in 12 bits).

Рисунок 27-7 показывает блок-схема для LZW сжатия. Таблица 27-3 обеспечивает постепенные подробности для примера входного файла, состоящего из 45 байтов, строка текста ASCII: *the/rain/in/Spain/falls/mainly/on/the/plain*. Когда мы говорим, что алгоритм LZW читает символ "a" от входного файла, мы подразумеваем, что это читает значение: 01100001 (97 выраженный в 8 битах), где 97 - "a" в ASCII. Когда мы говорим, что это записывает символ "a" к закодированному файлу, мы подразумеваем, что это записывает: 000001100001 (97 выраженный в 12 битах).

The compression algorithm uses two variables: *CHAR* and *STRING*. The variable, *CHAR*, holds a single character, i.e., a single byte value between 0 and 255. The variable, *STRING*, is a variable length string, i.e., a group of one or more characters, with each character being a single byte. In box 1 of Fig. 27-7, the program starts by taking the first byte from the input file, and placing it in the variable, *STRING*. Table 27-3 shows this action in line 1. This is followed by the algorithm looping for each additional byte in the input file, controlled in the flow diagram by box 8. Each time a byte is read from the input file (box 2), it is stored in the variable, *CHAR*. The data table is then searched to determine if the concatenation of the two variables, *STRING+CHAR*, has already been assigned a code (box 3).

Алгоритм компрессии использует две переменные: *CHAR* и *STRING*. Переменная, *STRING*, проводит(держит) одиночный буквенно-цифровой символ, то есть, единственное(отдельное) значение байта между 0 и 255. Переменная, *CHAR*, является строкой переменной длины, то есть, группа одного или большего количества символов, с каждым символом, являющимся единственным(отдельным) байтом. В поле 1 рис. 27-7, запуски программы, беря первый байт от входного файла, и помещая это в переменной, *STRING*. Таблица 27-3 показывает это действие в строке 1. Это сопровождается выполнением цикла алгоритма для каждого дополнительного байта во входном файле, управляется в блок-схеме полем 8. Каждый раз байт читается от входного файла (поле 2), это сохранено в переменной, *CHAR*. Таблица данных тогда обыскана, чтобы определить, если конкатенация из этих двух переменных, STRING+CHAR, уже была назначена код (поле 3).

If a match in the code table is *not* found, three actions are taken, as shown in boxes 4, 5 & 6. In box 4, the 12 bit code corresponding to the contents of the variable, *STRING*, is written to the compressed file. In box 5, a new code is created in the table for the concatenation of *STRING+CHAR*. In box 6, the variable, *STRING*, takes the value of the variable, *CHAR*. An example of these actions is shown in lines 2 through 10 in Table 27-3, for the first 10 bytes of the example file.

Если соответствие в таблице кода не найдено, три действия приняты, как показано в полях 4, 5 и 6. В поле 4, код 12 двоичных разрядов, соответствующих содержанию переменной, *STRING*, написана к сжатому файлу. В поле 5, новый код создан в таблице для конкатенации *STRING+CHAR*. В поле 6, переменная, *STRING*, принимает значение переменной, *CHAR*. Пример этих действий показывается в строках 2 до 10 в таблице 27-3, для примера первых 10 байтов файла.

	CHAR	*STRING + CHAR*	In Table?	Output	Add to Table	New *STRING*	Comments
1	t	t				t	first character- no action
2	h	th	no	t	256 = th	h	
3	e	he	no	h	257 = he	e	
4	/	e/	no	e	258 = e/	/	
5	r	/r	no	/	259 = /r	r	
6	a	ra	no	r	260 = ra	a	
7	i	ai	no	a	261 = ai	i	
8	n	in	no	i	262 = in	n	
9	/	n/	no	n	263 = n/	/	
10	i	/i	no	/	264 = /i	i	
11	n	in	yes (262)			in	first match found
12	/	in/	no	262	265 = in/	/	
13	S	/S	no	/	266 = /S	S	
14	p	Sp	no	S	267 = Sp	p	
15	a	pa	no	p	268 = pa	a	
16	i	ai	yes (261)			ai	matches *ai*, *ain* not in table yet
17	n	ain	no	261	269 = ain	n	*ain* added to table
18	/	n/	yes (263)			n/	
19	f	n/f	no	263	270 = n/f	f	
20	a	fa	no	f	271 = fa	a	
21	l	al	no	a	272 = al	l	
22	l	ll	no	l	273 = ll	l	
23	s	ls	no	l	274 = ls	s	
24	/	s/	no	s	275 = s/	/	
25	m	/m	no	/	276 = /m	m	
26	a	ma	no	m	277 = ma	a	
27	i	ai	yes (261)			ai	matches *ai*
28	n	ain	yes (269)			ain	matches longer string, *ain*
29	l	ainl	no	269	278 = ainl	l	
30	y	ly	no	l	279 = ly	y	
31	/	y/	no	y	280 = y/	/	
32	o	/o	no	/	281 = /o	o	
33	n	on	no	o	282 = on	n	
34	/	n/	yes (263)			n/	
35	t	n/t	no	263	283 = n/t	t	
36	h	th	yes (256)			th	matches *th*, *the* not in table yet
37	e	the	no	256	284 = the	e	*the* added to table
38	/	e/	yes			e/	
39	p	e/p	no	258	285 = e/p	p	
40	l	pl	no	p	286 = pl	l	
41	a	la	no	l	287 = la	a	
42	i	ai	yes (261)			ai	matches *ai*
43	n	ain	yes (269)			ain	matches longer string *ain*
44	/	ain/	no	269	288 = ain/	/	
45	EOF	/		/			end of file, output *STRING*

TABLE 27-3
LZW example. This shows the compression of the phrase: *the/rain/in/Spain/falls/mainly/on/the/plain/*.

When a match in the code table *is* found (box 3), the concatenation of *STRING+CHAR* is stored in the variable, *STRING*, without any other action taking place (box 7). That is, if a matching sequence is found in the table, no action should be taken before determining if there is a *longer* matching sequence also in the table. An example of this is shown in line 11, where the sequence: *STRING+CHAR = in*, is identified as already having a code in the table. In line 12, the next character from the input file, /, is added to the sequence, and the code table is searched for: *in/*.

Since this longer sequence is not in the table, the program *adds* it to the table, outputs the code for the shorter sequence that *is* in the table (code 262), and starts over searching for sequences beginning with the character, /. This flow of events is continued until there are no more characters in the input file. The program is wrapped up with the code corresponding to the current value of *STRING* being written to the compressed file (as illustrated in box 9 of Fig. 27-7 and line 45 of Table 27-3).

Когда в таблице соответствие кода *is* найдено (поле 3), конкатенация *STRING+CHAR* сохранена в переменной, *STRING*, без любого другого действия, имеющего место (поле 7). То есть если последовательность соответствия найдена в таблице, никакое действие не должно быть принято перед определением, если там имеется *longer* Соответствие последовательности также в таблице. Пример этого показывается в строке 11, где последовательность: *STRING+CHAR = in*, идентифицирован как уже наличие кода в таблице. В строке 12, следующий символ от входного файла, /, добавлен к последовательности, и таблица кода разыскивается: в *in*/. Так как эта более длинная последовательность не в таблице, программа прибавляет это к таблице, выходы кода для более короткой последовательности, которая находится в таблице (чтобы закодировать 262), и запусках по поиску последовательностей, начинающихся с символа, /. Этот поток событий продолжен до не имеется больше символов во входном файле. Программа обернута с кодом, соответствующим текущему значению *STRING*, записываемой в сжатый файл (как иллюстрировано в поле 9 рис. 27-7 и строки 45 из таблицы 27-3).

A flowchart of the LZW uncompression algorithm is shown in Fig. 27-8. Each code is read from the compressed file and compared to the code table to provide the translation. As each code is processed in this manner, the code table is updated so that it continually matches the one used during the compression. However, there is a small complication in the uncompression routine. There are certain combinations of data that result in the uncompression algorithm receiving a code that does not yet exist in its code table. This contingency is handled in boxes 4,5 & 6.

Блок-схема LZW некомпрессионного алгоритма показывается в рис. 27-8. Каждый код читается от сжатого файла и сравнен к таблице кода, чтобы обеспечить трансляцию. Поскольку каждый код обработан этим способом, таблица кода модифицирована так, чтобы это непрерывно соответствовало тому, используемому в течение сжатия. Однако, имеется маленькое осложнение в некомпрессионной подпрограмме. Имеются некоторые комбинации данных, которые приводят к некомпрессионному алгоритму, получающему код, который еще не существует в его таблице кода. Эта сопряженность признаков обработана в полях 4,5 и 6.

Only a few dozen lines of code are required for the most elementary LZW programs. The real difficulty lies in the efficient management of the code table. The brute force approach results in large memory requirements and a slow program execution. Several tricks are used in commercial LZW programs to improve their performance. For instance, the memory problem arises because it is not know beforehand how long each of the character strings for each code will be. Most LZW programs handle this by taking advantage of the redundant nature of the code table. For example, look at line 29 in Table 27-3, where code 278 is defined to be *ainl*. Rather than storing these four bytes, code 278 could be stored as: *code 269 + l*, where code 269 was previously defined as *ain* in line 17. Likewise, code 269 would be stored as: *code 261 + n*, where code 261 was previously defined as *ai* in line 7. This pattern always holds: every code can be expressed as a previous code plus one new character.

Только несколько строк программы обязательны для наиболее элементарных программ LZW. Реальная трудность находится в эффективном управлении таблицей кода. Подход

решения "в лоб" приводит к требованиям памяти большой емкости и медленному выполнению программы. Несколько уловок используются в коммерческих программах LZW, чтобы улучшить их эффективность. Например, проблема памяти возникает, потому что это - не, знают заранее, как длинная каждая из символьных строк для каждого кода будет. Большинство программ LZW обрабатывает это, пользуясь преимуществом избыточным характером таблицы кода. Например, смотрите на строку 29 в таблице 27-3, где код 278 определен, чтобы быть *ainl*. Скорее чем сохранение этих четырех байтов, код 278, мог быть сохранен как: *code 269 + l*, где код 269 был предварительно определен как *ain* в строке 17. Аналогично, код 269 был бы сохранен как: *code 261 + n*, где код 261 был предварительно определен как *ai* в строке 7. Этот образец всегда держится: Каждый код может быть выражен как предыдущий код плюс один новый символ.

The execution time of the compression algorithm is limited by searching the code table to determine if a match is present. As an analogy, imagine you want to find if a friend's name is listed in the telephone directory. The catch is, the only directory you have is arranged by telephone number, not alphabetical order. This requires you to search page after page trying to find the name you want. This inefficient situation is exactly the same as searching all 4096 codes for a match to a specific character string. The answer: organize the code table so that what you are looking for tells you where to look (like a partially alphabetized telephone directory). In other words, don't assign the 4096 codes to sequential locations in memory. Rather, divide the memory into sections based on what sequences will be stored there. For example, suppose we want to find if the sequence: *code 329 + x,* is in the code table. The code table should be organized so that the "*x*" indicates where to starting looking. There are many schemes for this type of code table management, and they can become quite complicated.

Время выполнения алгоритма компрессии ограничено, ища таблицу кода, чтобы определить, присутствует ли соответствие. Как аналогия, вообразите, что Вы хотите найти, перечислено ли имя друга в телефонном справочнике. Арретир, единственный каталог, который Вы имеете, размещается в порядке телефонных номеров, не алфавитный порядок. Это требует, чтобы Вы искали страницу за страницей, пробуя найти имя, которое Вы хотите. Это неэффективное положение(ситуация) - точно тот же самое как поиск всех 4096 кодов для соответствия специфической символьной строке. Ответ: организуйте таблицу кода так, чтобы, что Вы искали, сообщает Вам, где смотреть (подобно частично упорядочиваемому по алфавиту телефонному справочнику). Другими словами, не назначите 4096 кодов на последовательные расположения в памяти. Скорее, делите память в разделы, основанные на том, что последовательности будут сохранены там. Например, предположите, что мы хотим найти последовательность: *code 329 + x*, если она находится в таблице кода. Таблица кода должна быть организована так, чтобы "*x*" указал, где начинать просмотр. Имеются много схем этого типа управления таблицей кода, и они могут стать весьма сложными.

This brings up the last comment on LZW and similar compression schemes: *it is a very competitive field*. While the basics of data compression are relatively simple, the kinds of programs sold as commercial products are extremely sophisticated. Companies make money by selling you programs that perform compression, and jealously protect their trade-secrets through patents and the like. Don't expect to achieve the same level of performance as these programs in a few hours work.

Это поднимает последний комментарий относительно LZW и подобных компрессионных схем: это - очень конкурентоспособное поле. В то время как основы сжатия данных относительно просты, виды проданных программ, поскольку коммерческие программы чрезвычайно сложны. Компании делают деньги, продавая, Вам программы, которые испол-

няют сжатие, и ревниво защищают их торговые секреты через патенты и т.п. Не ожидайте достичь того же самого уровня эффективности как эти программы, поработав несколько часов.

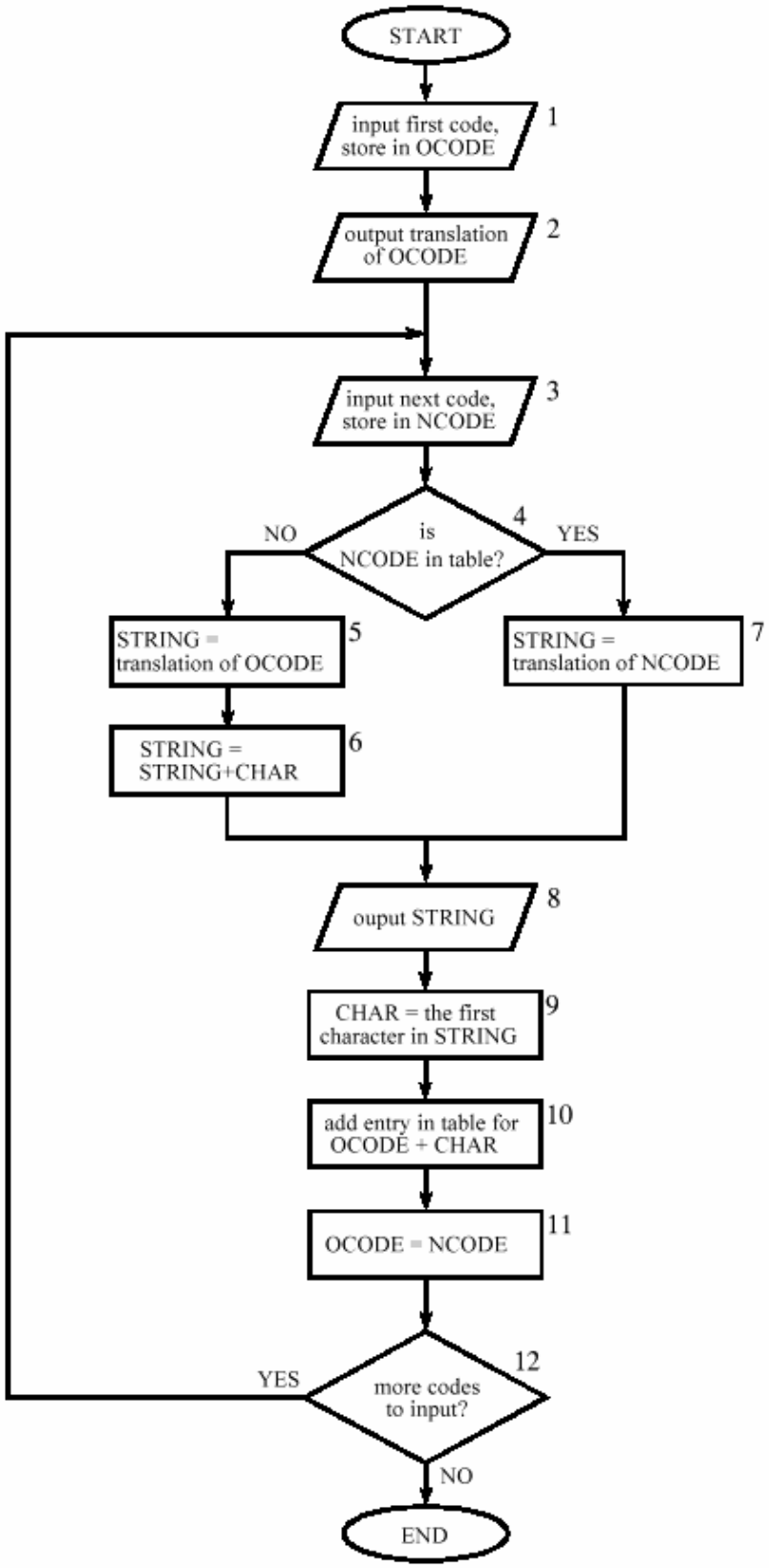


FIGURE 27-8
LZW uncompression flowchart. The variables, *OCODE* and *NCODE* (oldcode and newcode), hold the 12 bit codes from the compressed file, *CHAR* holds a single byte, *STRING* holds a string of bytes.
ЧИСЛО(РИСУНОК) 27-8

Блок-схема некомпрессионного LZW. Переменные, *OCODE* и *NCODE* (старый код и новый код), проводят(держат) коды 12 двоичных разрядов от сжатого файла, *CHAR* проводит(держит) единственный(отдельный) байт, *STRING* проводит(держит) строку байтов.

JPEG (Transform Compression)
JPEG (Преобразовывают Сжатие)

Many methods of lossy compression have been developed; however, a family of techniques called *transform compression* has proven the most valuable. The best example of transform compression is embodied in the popular JPEG standard of image encoding. JPEG is named after its origin, the *Joint Photographers Experts Group*. We will describe the operation of JPEG to illustrate how lossy compression works.

Много методов сжатия с потерями(lossy) были разработаны; однако, семейство методов называемых *сжатием трансформанты* доказало наибольшую ценность. Лучший пример сжатия трансформанты воплощен в популярном стандарте JPEG кодирования изображения. JPEG назван по имени его происхождения, *Joint Photographers Experts Group* (Совместная Группа Экспертов Фотографов). Мы опишем операцию JPEG, чтобы иллюстрировать, как lossy сжатие(сжатие с потерями) работает.

We have already discussed a simple method of lossy data compression, *coarser sampling and/or quantization* (CS&Q in Table 27-1). This involves reducing the number of bits per sample or entirely discard some of the samples. Both these procedures have the desired effect: the data file becomes smaller at the expense of signal quality. As you might expect, these simple methods do not work very well.

Мы уже обсудили простой метод lossy сжатия данных, *более крупного осуществления выборки и/или квантования* (CS&Q в таблице 27-1). Это включает в себя(подразумевает) сокращение номера(числа) битов на выборку или полностью отказываться от некоторых из выборок. Обе эти процедуры имеют желательный эффект: файл данных становится меньшим за счет качества сигнала. Поскольку Вы могли бы ожидать, эти простые методы работают не очень хорошо.

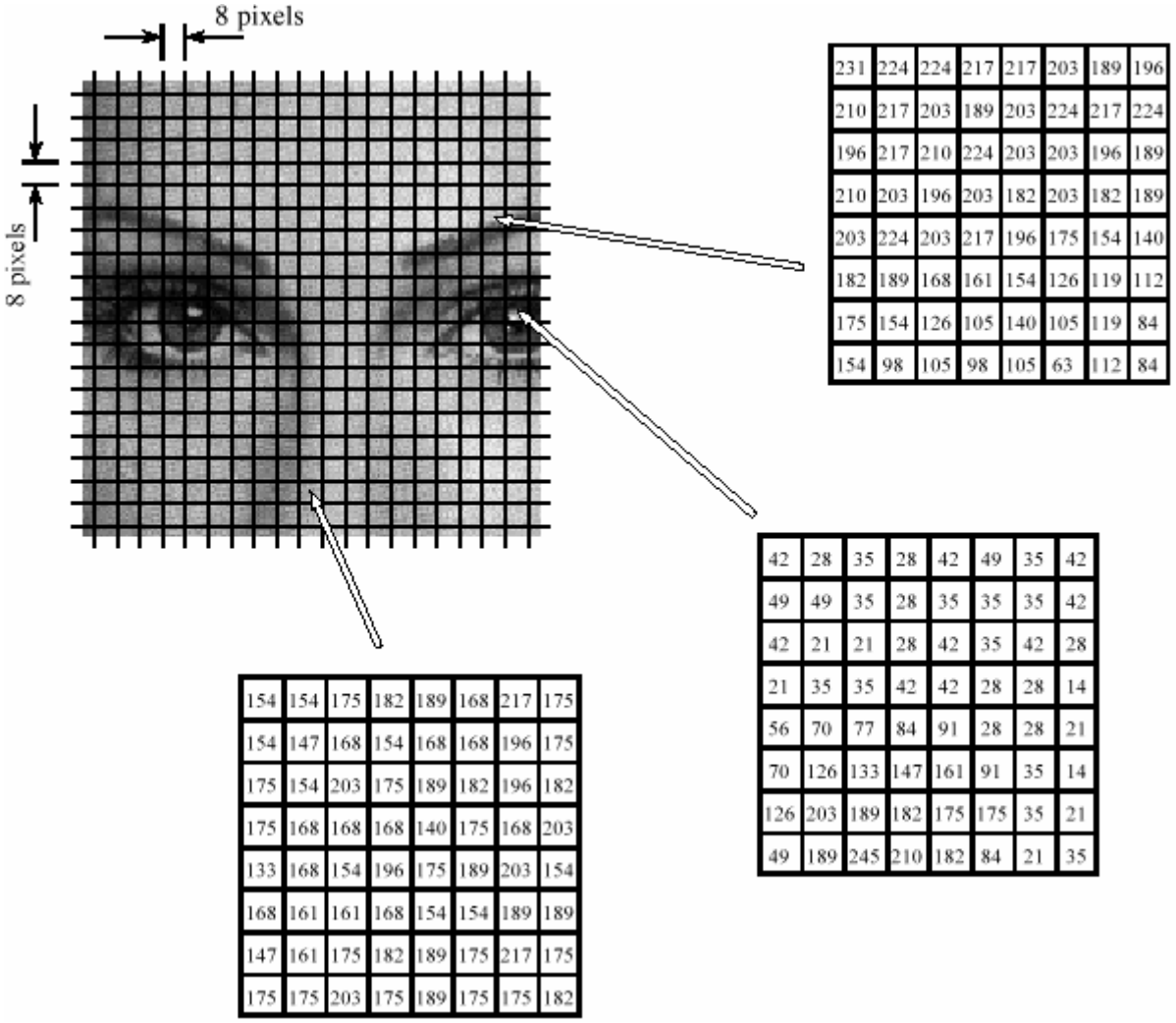


FIGURE 27-9
JPEG image division. JPEG transform compression starts by breaking the image into 8x8 groups, each containing 64 pixels. Three of these 8x8 groups are enlarged in this figure, showing the values of the individual pixels, a single byte value between 0 and 255.

РИСУНОК 27-9
Разделение изображений JPEG. JPEG начинает преобразование сжатием, разбивая изображение на группы 8x8, каждая из которых содержит 64 пиксела. Три из этих 8x8 групп расширены(увеличены в масштабе) в этом рисунке, показывая значения индивидуальных пикселов, единственное значение байта между 0 и 255.

Transform compression is based on a simple premise: when the signal is passed through the Fourier (or other) transform, the resulting data values will no longer be equal in their information carrying roles. In particular, the low frequency components of a signal are more important than the high frequency components. Removing 50% of the bits from the high frequency components might remove, say, only 5% of the encoded information.

Сжатие Трансформанты основано на простой предпосылке: когда сигнал пропускают через трансформанту Фурье (или другую) трансформанту, полученные значения данных больше не будет равны в их роли несения информации. В частности низкочастотные компоненты сигнала, более важны, чем компоненты высокой частоты. Удаление 50% битов компонентов высокой частоты могло бы удалять, скажем, только 5% закодированной информации.

As shown in Fig. 27-9, JPEG compression starts by breaking the image into 8x8 pixel groups. The full JPEG algorithm can accept a wide range of bits per pixel, including the use of color information. In this example, each pixel is a single byte, a grayscale value between 0 and 255. These 8x8 pixel groups are treated independently during compression. That is, each group is initially represented by 64 bytes. After transforming and removing data, each group is represented by, say, 2 to 20 bytes. During uncompression, the inverse transform is taken of the 2 to 20 bytes to create an approximation of the original 8x8 group. These approximated groups are then fitted together to form the uncompressed image. Why use 8x8 pixel groups instead of, for instance, 16x16? The 8x8 grouping was based on the maximum size that integrated circuit technology could handle at the time the standard was developed. In any event, the 8Ч8 size works well, and it may or may not be changed in the future.

Как показано на рис. 27-9, JPEG начинает сжатие разбивая изображение на группы пикселов 8x8. Полный алгоритм JPEG может принимать широкий диапазон битов на пиксел, включая использование цветной информации. В этом примере, каждый пиксел - единственный(отдельный) байт, полутоновое значение между 0 и 255. Эти группы 8x8 пикселов в течение сжатия обработаны независимо. То есть каждая группа первоначально представлена 64 байтами. После преобразования и удаления данных, каждая группа представлена, скажем, от 2 до 20 байтов. В течение несжатия, обратная трансформанта принята из от 2 до 20 байтов, чтобы создать аппроксимацию оригинала группы 8x8. Эти аппроксимированные группы тогда приспособлены вместе, чтобы формировать несжатое изображение. Почему используются группы пикселов 8x8 вместо, например, 16x16? Группировка 8x8 была основана на максимальном размере, который технология интегральной схемы могла обрабатывать во время, когда стандарт разрабатывался. В любом случае, размер группы 8x8 работает хорошо, и это может или не может быть изменено в будущем.

Many different transforms have been investigated for data compression, some of them invented specifically for this purpose. For instance, the *Karhunen-Loeve* transform provides the best possible compression ratio, but is difficult to implement. The *Fourier transform* is easy to use, but does not provide adequate compression. After much competition, the winner is a relative of the Fourier transform, the **Discrete Cosine Transform (DCT)**.

Много различных трансформант были исследованы для сжатия данных, некоторые из них изобретенный определенно для этой цели. Например, трансформанта *Karhunen-Loeve(*разложение *Карунена Лозва)* обеспечивает возможно лучшее компрессионное отношение(коэффициент), но трудна в осуществлении. Преобразование Фурье(трансформанта Фурье) удобно, но не обеспечивает адекватное сжатие. После долгого соревнования, победитель -, **Дискретная Трансформанта Косинуса** (**ДТК**), из семейства преобразований Фурье.

Just as the Fourier transform uses sine and cosine waves to represent a signal, the DCT only uses cosine waves. There are several versions of the DCT, with slight differences in their mathematics. As an example of one version, imagine a 129 point signal, running from sample 0 to sample 128. Now, make this a 256 point signal by duplicating samples 1 through 127 and adding them as samples 255 to 130. That is: Taking the 0, 1, 2, …, 127, 128, 127, …, 2, 1. Fourier transform of this 256 point signal results in a frequency spectrum of 129 points, spread between 0 and 128. Since the time domain signal was forced to be symmetrical, the spectrum's imaginary part will be composed of all zeros. In other words, we started with a 129 point time domain signal, and ended with a frequency spectrum of 129 points, each the amplitude of a cosine wave. Voila, the DCT!

Так же, как трансформанта Фурье использует волны синуса и косинуса, чтобы представить сигнал, ДТК использует только волны косинуса. Имеются несколько версий ДТК, с небольшими различиями в их математике. Как пример одной версии, вообразите 129 точек сигнала, выполняющиеся от выборки 0, до выборки 128. Теперь, делайте(возьмите) эти 256 точек сигнала, дублируя выборки от 1 до 127 и прибавляя их как выборки от 255 до 130. Это: Беря(Делая) 0, 1, 2, …, 127, 128, 127, …, 2, 1. Трансформанта Фурье из этих 256 точек сигнала приводит к спектру частот 129 точек, распространенных между 0 и 128. Так как сигнал домена времени был вынужден быть симметрическим, мнимая часть спектра будет составлена из всех нулей. Другими словами, мы начали со 129 точек сигнала домена времени, и закончили спектром частот 129 точек, каждый амплитуда волны косинуса. Voila, ДТК!

When the DCT is taken of an 8x8 group, it results in an 8x8 spectrum. In other words, 64 numbers are changed into 64 other numbers. All these values are *real*; there is no complex mathematics here. Just as in Fourier analysis, each value in the spectrum is the amplitude of a **basis function**. Figure 27-10 shows 6 of the 64 basis functions used in an 8x8 DCT, according to where the amplitude sits in the spectrum. The 8x8 DCT basis functions are given by:

Когда ДТК группы 8x8 принят, это приводит к спектру 8x8. Другими словами, 64 номера заменены на 64 других номера. Все эти значения реальные; не имеется никакой комплексной математики здесь. Также, как в анализе Фурье, каждое значение в спектре - амплитуда **базисной функции**. Рисунок 27-10 показывает 6 из 64 базисных функций, используемых в ДТК 8x8, согласно тому, где амплитуда находится в спектре. базисная функция ДТК 8x8 дается:

EQUATION 27-1

DCT basis functions. The variables x & y are the indexes in the spatial domain, and u & v are the indexes in the frequency spectrum. This is for an 8x8 DCT, making all the indexes run from 0 to 7.

$$b[x,y] = \cos\left[\frac{(2x+1)u\pi}{16}\right]\cos\left[\frac{(2y+1)v\pi}{16}\right]$$

УРАВНЕНИЕ 27-1

Базисные функция ДТК. Переменные x и y - индексы в пространственном домене, и u, и v - индексы в спектре частот. Это - для ДТК 8x8, делая все индексы, выполненные от 0 до 7.

The low frequencies reside in the upper-left corner of the spectrum, while the high frequencies are in the lower-right. The DC component is at [0,0], the upper-left most value. The basis function for [0,1] is one-half cycle of a cosine wave in one direction, and a constant value in the other. The basis function for [1,0] is similar, just rotated by 90º.

Низкие частоты постоянно находятся в левом верхнем углу спектра, в то время как высокие частоты находятся в нижнем правом. Компонент постоянного тока - в [0,0], левое верхнее большинство значения. Базисная функция для [0,1] - половина периода волны косинуса в одном направлении, и постоянном значении в другом. Базисная функция для [1,0] подобна, только повернута(смещена) на 90º.

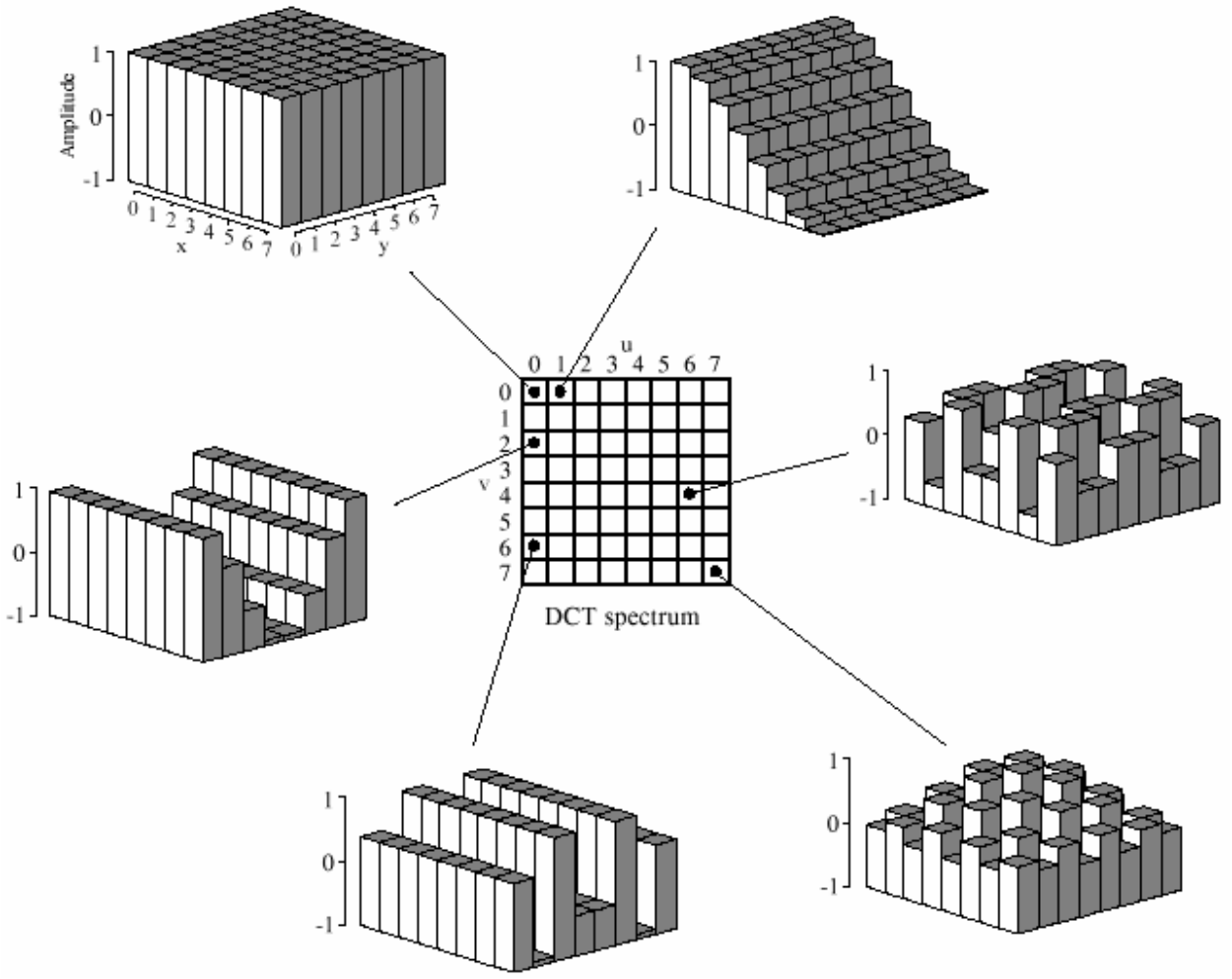


FIGURE 27-10
The DCT basis functions. The DCT spectrum consists of an 8x8 array, with each element in the array being an amplitude of one of the 64 basis functions. Six of these basis functions are shown here, referenced to where the corresponding amplitude resides.
РИСУНОК 27-10
Базисные функции ДТК. Спектр ДТК состоит из массива 8x8, с каждым элементом в массиве, являющемся амплитудой одной из этих 64 базисных функций. Шесть из этих базисных функций показываются здесь, упомянуты к тому, где соответствующая амплитуда постоянно находится.

The DCT calculates the spectrum by *correlating* the 8x8 pixel group with each of the basis functions. That is, each spectral value is found by multiplying the appropriate basis function by the 8Ч8 pixel group, and then summing the products. Two adjustments are then needed to finish the DCT calculation (just as with the Fourier transform). First, divide the 15 spectral values in row 0 and column 0 by *two*. Second, divide all 64 values in the spectrum by 16. The inverse DCT is calculated by assigning each of the amplitudes in the spectrum to the proper basis function, and summing to recreate the spatial domain. No extra steps are required. These are exactly the same concepts as in Fourier analysis, just with different basis functions.

ДТК вычисляет спектр, коррелируя группу пикселов 8x8 с каждой из базисных функций. То есть каждое спектральное значение найдено, умножая соответствующую базисную функция на группу пикселов 8x8, и затем сумируя продукты. Две корректировки тогда необходимы, чтобы закончить вычисление ДТК (также, как с преобразованием Фурье). Во первых, делите 15 спектральных значений в строке 0 и столбце 0 два. Во вторых, делите все 64 значения в спектре 16. Обратный ДТК рассчитан, назначая каждую из амплитуд в

спектре к надлежащей базисной функция, и подводя итог освежить(пересоздать) пространственный домен. Никакие дополнительные шаги не требуются. Они - точно те же самые концепции как в анализе Фурье, только с различными базисными функция.

Figure 27-11 illustrates JPEG encoding for the three 8x8 groups identified in Fig. 27-9. The left column, Figs. a, b & c, show the original pixel values. The center column, Figs. d, e & f, show the DCT spectra of these groups.

Рисунок 27-11 иллюстрирует кодирование JPEG для трех групп 8x8, идентифицированных на рис. 27-9. Левый столбец, рис. a, b и c, показывает первоначальные значения пиксела. Средний столбец, рис. d, e и f, показывает спектры ДТК этих групп.

Original Group	DCT Spectrum	Quantization Error

a. Eyebrow

231	224	224	217	217	203	189	196
210	217	203	189	203	224	217	224
196	217	210	224	203	203	196	189
210	203	196	203	182	203	182	189
203	224	203	217	196	175	154	140
182	189	168	161	154	126	119	112
175	154	126	105	140	105	119	84
154	98	105	98	105	63	112	84

d. Eyebrow spectrum

174	19	0	3	1	0	-3	1
52	-13	-3	-4	-4	-4	5	-8
-18	-4	8	3	3	2	0	9
5	12	-4	0	0	-5	-1	0
1	2	-2	-1	4	4	2	0
-1	2	1	3	0	0	1	1
-2	5	-5	-5	3	2	-1	-1
3	5	-7	0	0	0	-4	0

g. Using 10 bits

0	0	0	0	-1	0	0	0
-1	0	0	0	0	0	0	-1
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	1	0	0	0	-1	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

b. Eye

42	28	35	28	42	49	35	42
49	49	35	28	35	35	35	42
42	21	21	28	42	35	42	28
21	35	35	42	42	28	28	14
56	70	77	84	91	28	28	21
70	126	133	147	161	91	35	14
126	203	189	182	175	175	35	21
49	189	245	210	182	84	21	35

e. Eye spectrum

70	24	-28	-4	-2	-10	-1	0
-53	-35	43	13	7	13	1	3
23	9	-10	-8	-7	-6	5	-3
6	2	-2	8	2	-1	0	-1
-10	-2	-1	-12	2	1	-1	4
3	0	0	11	-4	-1	5	6
-3	-5	-5	-4	3	2	-3	5
3	0	4	5	1	2	1	0

h. Using 8 bits

0	-3	-1	-1	1	0	0	-1
1	0	-1	-1	0	0	0	-1
-1	-2	1	0	-2	0	-2	-2
-1	-2	-1	2	0	2	0	1
0	-2	1	0	0	1	0	0
0	-4	-1	0	1	0	0	0
0	-2	0	1	-1	-1	1	-1
-1	-3	1	1	1	-3	-2	-1

c. Nose

154	154	175	182	189	168	217	175
154	147	168	154	168	168	196	175
175	154	203	175	189	182	196	182
175	168	168	168	140	175	168	203
133	168	154	196	175	189	203	154
168	161	161	168	154	154	189	189
147	161	175	182	189	175	217	175
175	175	203	175	189	175	175	182

f. Nose spectrum

174	-11	-2	-3	-3	6	-3	4
-2	-3	1	2	0	3	1	2
3	0	-4	0	0	0	-1	9
-4	-6	-2	1	-1	4	-10	-3
1	2	-2	0	0	-2	0	-5
3	-1	3	-2	2	1	1	0
3	5	2	-2	3	0	4	3
4	-3	-13	3	-4	3	-5	3

i. Using 5 bits

-13	-7	1	4	0	0	10	-2
-22	6	-13	5	-5	2	-2	-13
-9	-15	0	-17	-8	8	12	25
-9	16	1	9	1	-5	-5	13
-20	-3	-13	-16	-19	-1	-4	-22
-11	6	-8	16	-9	-3	-7	6
-14	10	-9	4	-15	3	3	-4
-13	19	12	9	18	5	-5	10

FIGURE 27-11
Example of JPEG encoding. The left column shows three 8x8 pixel groups, the same ones shown in Fig. 27-9. The center column shows the DCT spectra of these three groups. The third column shows the error in the uncompressed pixel values resulting from using a finite number of bits to represent the spectrum.
РИСУНОК 27-11
Пример кодирования JPEG. Левый столбец показывает три группы пикселов 8x8, те же самые, что показаны в рис. 27-9. Средний столбец показывает спектры ДТК из этих трех групп. Третий столбец показывает

ошибку в несжатых значениях пиксела, следующих из использования конечного числа битов, чтобы представить спектр.

The right column, Figs. g, h & i, shows the effect of reducing the number of bits used to represent each component in the frequency spectrum. For instance, (g) is formed by truncating each of the samples in (d) to ten bits, taking the inverse DCT, and then subtracting the reconstructed image from the original. Likewise, (h) and (i) are formed by truncating each sample in the spectrum to eight and five bits, respectively. As expected, the error in the reconstruction increases as fewer bits are used to represent the data. As an example of this bit truncation, the spectra shown in the center column are represented with 8 bits per spectral value, arranged as 0 to 255 for the DC component, and –127 to 127 for the other values.

Правый столбец, рис. g, h и i, показывает, что эффект сокращения числа битов обычно используемый для представления каждого компонента в спектре частот. Например, (g) сформирован, усекая каждую из выборок в (d) к десяти битам, беря обратный ДТК, и затем вычитая восстановленное изображение от оригинала. Аналогично, (h) и (i) сформированы, усекая каждую выборку в спектре к восьми и пяти битам, соответственно. Столь же ожидаема, ошибка в увеличениях реконструкции, как меньшее количество битов используется, чтобы представить данные. Как пример этого усечения двоичных разрядов, спектры, показанные в среднем столбце представлены 8-ю битами на значение спектра, размещается как от 0 до 255 для компонента постоянного тока, и от -127 до 127 для других значений.

The second method of compressing the frequency domain is to discard some of the 64 spectral values. As shown by the spectra in Fig. 27-11, nearly all of the signal is contained in the low frequency components. This means the highest frequency components can be eliminated, while only degrading the signal a small amount. Figure 27-12 shows an example of the image distortion that occurs when various numbers of the high frequency components are deleted. The 8x8 group used in this example is the *eye* image of Fig. 27-10. Figure (d) shows the correct reconstruction using all 64 spectral values. The remaining figures show the reconstruction using the indicated number of lowest frequency coefficients. As illustrated in (c), even removing three-fourths of the highest frequency components produces little error in the reconstruction. Even better, the error that does occur looks very much like random noise.

Второй метод сжатия частотного домена состоит в том, чтобы отказаться от некоторых из 64 спектральных значений. Как показано спектрами в рис. 27-11, почти весь сигнал содержится в низких частотных компонентах. Это означает, что самые высокие частотные компоненты могут быть устранены, только при небольшом количестве ухудшения сигнала. Рисунок 27-12 показывает пример искажения изображения, которое происходит когда различный номера компонентов высокой частоты удалены. Группа 8x8, используемая в этом примере - изображение глаза на рис. 27-10. Рисунок (d) показывает правильную реконструкцию, используя все 64 спектральных значения. Остающиеся числа показывают реконструкцию, используя обозначенное число самых низких частотных коэффициентов. Как иллюстрировано в (c), даже удаление трех четвертей из самых высоких частотных компонентов производит небольшую ошибку в реконструкции. Даже лучше, ошибка, которая происходит выглядит очень подобно случайному шуму.

JPEG is good example of how several data compression schemes can be combined for greater effectiveness. The entire JPEG procedure is outlined in the following steps. First, the image is broken into the 8x8 groups. Second, the DCT is taken of each group. Third, each 8x8 spectrum is compressed by the above methods: reducing the number of bits and eliminating some of the components. This takes place in a single step, controlled by a **quantization table**. Two examples

of quantization tables are shown in Fig. 27-13. Each value in the spectrum is divided by the matching value in the quantization table, and the result rounded to the nearest integer. For instance, the upper-left value of the quantization table is *one*, resulting in the DC value being left unchanged. In comparison, the lower-right entry in (a) is 16, meaning that the original range of -127 to 127 is reduced to only -7 to 7. In other words, the value has been reduced in precision from eight bits to four bits. In a more extreme case, the lower-right entry in (b) is 256, completely eliminating the spectral value.

JPEG - хороший пример того, как несколько схем сжатия данных могут быть объединены для большей эффективности. Полная процедура JPEG выделена в следующих шагах. Во первых, изображение разбито на группы 8x8. Во вторых, принят ДТК каждой группы. Третье, каждый спектр группы 8x8 сжат вышеупомянутыми методами: сокращением числа битов и устранением некоторых из компонентов. Это имеет место в единственном(отдельном) шаге, управляемом **таблицей квантования**. Два примера таблиц квантования показываются на рис. 27-13. Каждое значение в спектре разделено значением соответствия в таблице квантования, и результату, округленном к самому близкому целому числу. Для образца, левое верхнее значение таблицы квантования *один*, приводя к значению постоянного тока, оставляемому неизменяемым. Для сравнения, нижний правый вход в (a) - 16, означая, что первоначальный диапазон от -127 до 127 сокращен к только от -7 до 7. Другими словами, значение было сокращено в прецизионности от восьми битов до четырех битов. В большем количестве критического случая, нижний правый вход в (b) - 256, полностью устраняя спектральное значение.

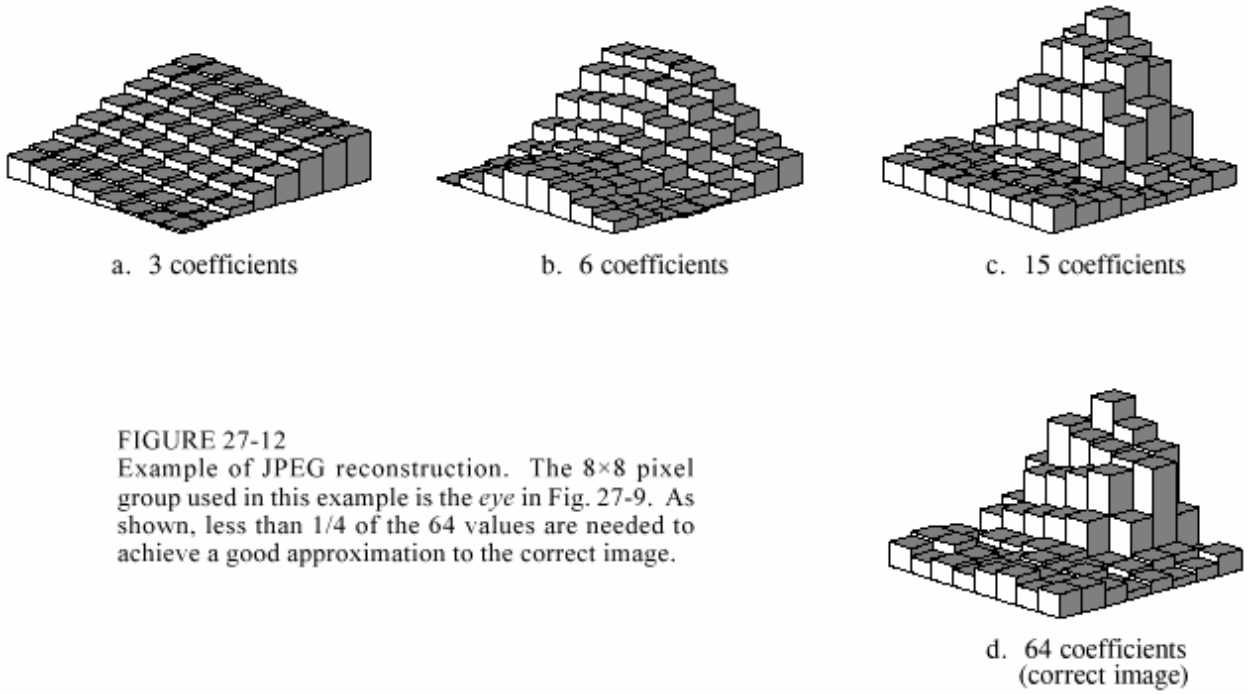


FIGURE 27-12
Example of JPEG reconstruction. The 8×8 pixel group used in this example is the *eye* in Fig. 27-9. As shown, less than 1/4 of the 64 values are needed to achieve a good approximation to the correct image.

FIGURE 27-12
Example of JPEG reconstruction. The 8x8 pixel group used in this example is the *eye* in Fig. 27-9. As shown, less than 1/4 of the 64 values are needed to achieve a good approximation to the correct image.

РИСУНОК 27-12
Пример JPEG реконструкции. группа пикселов 8x8, используемой в этом примере - *глаз* в рис. 27-9. Как показано, меньше чем 1/4 из 64 значений необходимы, чтобы достичь хорошей аппроксимации к правильному изображению.

a. Low compression

1	1	1	1	1	2	2	4
1	1	1	1	1	2	2	4
1	1	1	1	2	2	2	4
1	1	1	1	2	2	4	8
1	1	2	2	2	2	4	8
2	2	2	2	2	4	8	8
2	2	2	4	4	8	8	16
4	4	4	4	8	8	16	16

b. High compression

1	2	4	8	16	32	64	128
2	4	4	8	16	32	64	128
4	4	8	16	32	64	128	128
8	8	16	32	64	128	128	256
16	16	32	64	128	128	256	256
32	32	64	128	128	256	256	256
64	64	128	128	256	256	256	256
128	128	128	256	256	256	256	256

FIGURE 27-13
JPEG quantization tables. These are two example quantization tables that might be used during compression. Each value in the DCT spectrum is divided by the corresponding value in the quantization table, and the result rounded to the nearest integer.
РИСУНОК 27-13
Таблицы квантования JPEG. Они - две таблицы примера квантования, которые могли бы использоваться в течение сжатия. Каждое значение в спектре ДТК разделено соответствующим значением в таблице квантования, и результате, округленном к самому близкому целому числу.

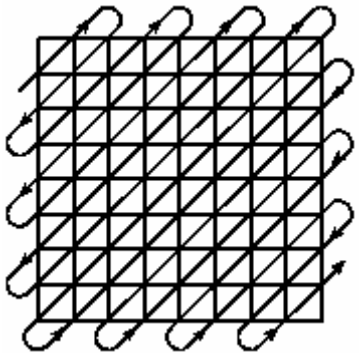
In the fourth step of JPEG encoding, the modified spectrum is converted from an 8x8 array into a linear sequence. The serpentine pattern shown in Figure 27-14 is used for this step, placing all of the high frequency components together at the end of the linear sequence. This groups the *zeros* from the eliminated components into long runs. The fifth step compresses these runs of zeros by run-length encoding. In the sixth step, the sequence is encoded by either Huffman or arithmetic encoding to form the final compressed file.

В четвертом шаге кодирования JPEG, изменяемый спектр преобразован из массива 8x8 в линейную последовательность. Змеевидный образец, показанный в Рисунке 27-14 используется для этого шага, помещая все компоненты высокой частоты вместе в конце линейной последовательности. Эти группы *нули* от устраненных компонентов в выполненной длине. Пятый шаг сжимает, эти выполненные нули кодированием с переменной длиной строки. В шестом шаге, последовательность закодирована или Хаффманом или арифметическим кодированием, чтобы формировать конечный сжатый файл.

The amount of compression, and the resulting loss of image quality, can be selected when the JPEG compression program is run. Figure 27-15 shows the type of image distortion resulting from high compression ratios. With the 45:1 compression ratio shown, each of the 8x8 groups is represented by only about 12 bits. Close inspection of this image shows that six of the lowest frequency basis functions are represented to some degree.

Количество сжатия, и потери качества заканчивающегося изображения, может быть отобрано, когда JPEG компрессионная программа выполнена. Рисунок 27-15 показывает тип искажения изображения, следующего из высоких компрессионных отношений(коэффициентов). С показанным компрессионным отношением 45:1, каждый из группы 8x8 представлен только приблизительно 12 битов. Близкий осмотр этого изображения показывает, что шесть из самых низких частотных базисных функций представлены до некоторой степени.

FIGURE 27-14
JPEG serial conversion. A serpentine pattern used to convert the 8x8 DCT spectrum into a linear sequence of 64 values. This places all of the high frequency components together, where the large number of zeros can be efficiently compressed with run-length encoding.
РИСУНОК 27-14
Последовательное преобразование JPEG. Змеевидный образец обычно используемый для преобразования спектра ДТК 8x8 в линейную последовательность 64 значений. Это размещает все компоненты высокой частоты вместе, где большое количество нулей может быть эффективно сжато кодированием с переменной длиной строки.



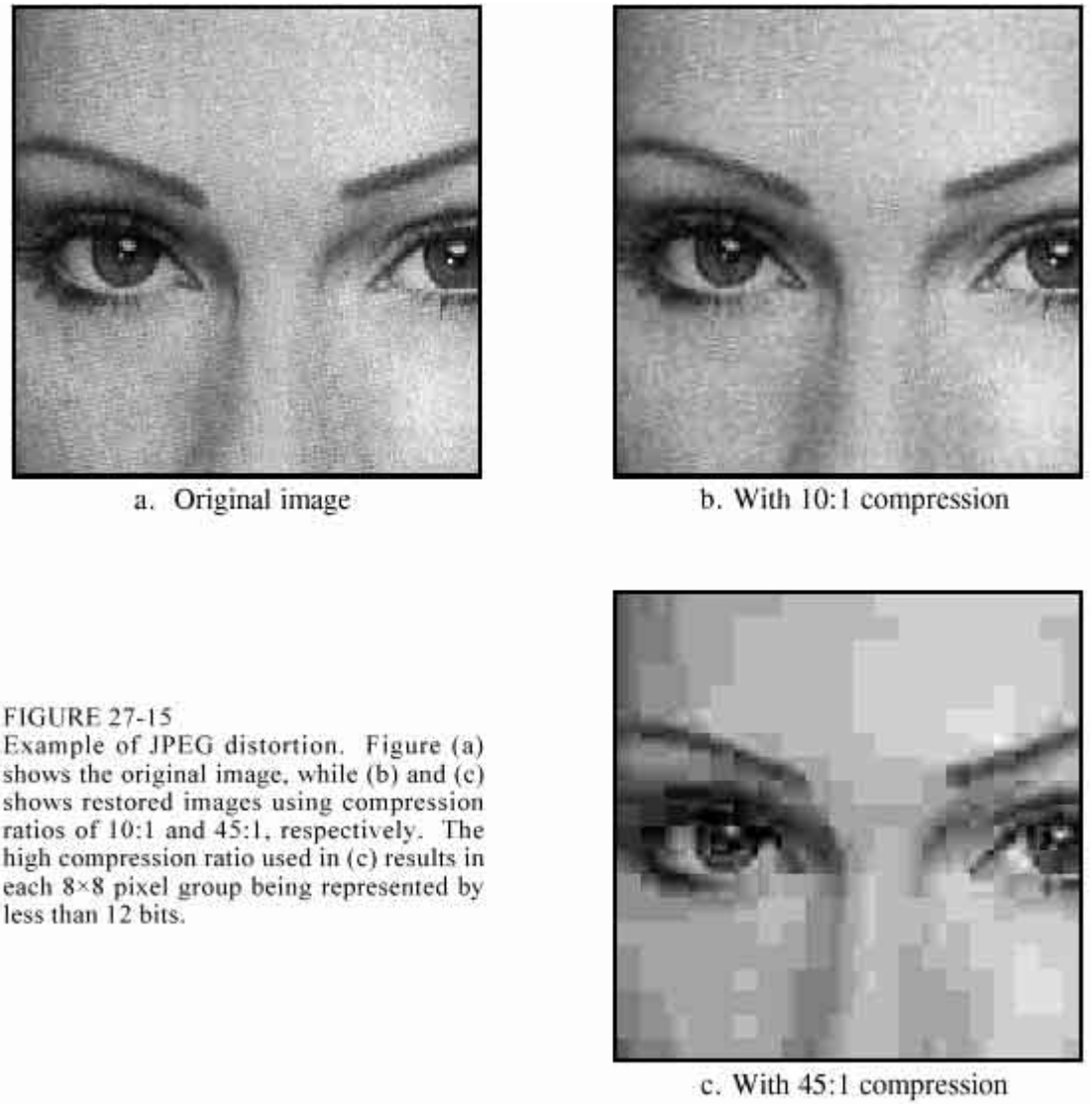


FIGURE 27-15
Example of JPEG distortion. Figure (a) shows the original image, while (b) and (c) shows restored images using compression ratios of 10:1 and 45:1, respectively. The high compression ratio used in (c) results in each 8×8 pixel group being represented by less than 12 bits.

FIGURE 27-15
Example of JPEG distortion. Figure (a) shows the original image, while (b) and (c) shows restored images using compression ratios of 10:1 and 45:1, respectively. The high compression ratio used in (c) results in each 8x8 pixel group being represented by less than 12 bits.

ЧИСЛО(РИСУНОК) 27-15

Пример искажения JPEG. Рисунок (a) показывает первоначальное изображение, в то время как (b) и (c) показывают восстановленные изображения, используя компрессионные отношения 10:1 и 45:1, соответственно. Высокое компрессионное отношение, используемое в (c) приводит к каждой группе пикселов 8x8, представляемой меньше чем 12 битами.

Why is the DCT better than the Fourier transform for image compression? The main reason is that the DCT has one-half cycle basis functions, i.e., S[0,1] and S[1,0]. As shown in Fig. 27-10, these gently slope from one side of the array to the other. In comparison, the lowest frequencies in the Fourier transform form *one complete cycle*. Images nearly always contain regions where the brightness is gradually changing over a region. Using a basis function that matches this basic pattern allows for better compression.

Почему - ДТК лучше чем преобразование Фурье для сжатия изображения? Основная причина - то, что ДТК имеет половину периода базисных функций, то есть, S[0,1] и S[1,0]. Как показано в рис. 27-10, они мягко имеют наклон от одной стороны массива к другой. Для сравнения, самые низкие частоты в преобразовании Фурье формируют *один полный период*. Изображение почти всегда содержит области, где яркость постепенно переключает область. Использование базисной функции, которая соответствует этому основному образцу, учитывает лучшее сжатие.

MPEG

MPEG is a compression standard for digital video sequences, such as used in computer video and digital television networks. In addition, MPEG also provides for the compression of the sound track associated with the video. The name comes from its originating organization, the *Moving Pictures Experts Group*. If you think JPEG is complicated, MPEG is a nightmare! MPEG is something you buy, not try to write yourself. The future of this technology is to encode the compression and uncompression algorithms directly into integrated circuits. The potential of MPEG is vast. Think of thousands of video channels being carried on a single optical fiber running into your home. This is a key technology of the 21st century.

MPEG - компрессионный стандарт для цифровых последовательностей видео, типа используемого в компьютерном видео и цифровых телевизионных сетях. Кроме того, MPEG также обеспечивает сжатие звуковой фонограммы, связанной с видео. Название спутник от его организации возникновения, *Moving Pictures Experts Group* (Экспертная Группа Перемещения Изображений). Если Вы думаете, что JPEG сложен, MPEG - кошмар! MPEG - кое-что, что Вы покупаете, не пробуют записать себя. Будущее этой технологии должно кодировать компрессионные и некомпрессионные алгоритмы непосредственно в интегральные схемы. Потенциал MPEG обширен. Думайте о тысячах видеоканалов, передаваемые по единственному световоду, к вашему дому. Это - ключевая технология 21-ого столетия.

In addition to reducing the data rate, MPEG has several important features. The movie can be played *forward* or in *reverse*, and at either *normal* or *fast* speed. The encoded information is *random access*, that is, any individual frame in the sequence can be easily displayed as a still picture. This goes along with making the movie *editable*, meaning that short segments from the movie can be encoded only with reference to themselves, not the entire sequence. MPEG is designed to be robust to errors. The last thing you want is for a single bit error to cause a disruption of the movie.

В дополнение к сокращению скорости передачи данных, MPEG имеет несколько важных особенностей. Кинофильм можно запускать *вперед* или *обратно*, или в нормальном или

ускоренном режиме. Закодированная информация - *произвольный доступ*, то есть любая индивидуальная структура(кадр) в последовательности может быть легко отображена как все еще изображение. Это идет наряду с созданием доступного для *редактирования* кинофильма, значение, что короткие сегменты от кинофильма могут быть закодированы только в отношении себя, не полная последовательность. MPEG разработан, чтобы быть устойчивым к ошибкам. Последняя вещь, которую Вы хотите - для единственной ошибки двоичных разрядов вызвать сбой кинофильма.

The approach used by MPEG can be divided into two types of compression: *within-the-frame* and *between-frame*. Within-the-frame compression means that individual frames making up the video sequence are encoded as if they were ordinary still images. This compression is preformed using the JPEG standard, with just a few variations. In MPEG terminology, a frame that has been encoded in this way is called an intra-coded or **I-picture**.

Подход, используемый MPEG может быть разделен на два типа сжатия: "в пределах структуры(рамки) и между структурой(рамкой)". Сжатие посредством "в пределах структуры(рамки)" что индивидуальные структуры(кадры), составляющие последовательность видео закодированы, как будто они были обычны все еще, изображения. Это сжатие - заготовка использования JPEG стандарта, с только несколько вариаций. В MPEG терминологии, структура(кадр), которая была закодирована таким образом, называется intra-coded(внутреннее кодирование) или **I-picture**.

Most of the pixels in a video sequence change very little from one frame to the next. Unless the camera is moving, most of the image is composed of a background that remains constant over dozens of frames. MPEG takes advantage of this with a sophisticated form of *delta encoding* to compress the redundant information *between frames*. After compressing one of the frames as an I-picture, MPEG encodes successive frames as predictive-coded or **P-pictures**. That is, only the pixels that have changed since the I-picture are included in the P-picture.

Большинство пикселов в последовательности видео изменяется очень немного от одного кадра до следующего. Если камера не перемещается, большая часть изображения составлена из фона, который остается константой более чем множества кадров. MPEG воспользуется преимуществом этого со сложной формой *дельта кодирования*, чтобы сжать избыточную информацию *между кадрами*. После сжатия одного из кадров как I-picture, MPEG кодирует последовательные структуры(кадры) как прогнозирующее кодирование или **P-pictures** (прогнозирование картинки?). То есть только пикселы, которые изменились начиная(после) I-picture включены в P-picture (P-изображение).

While these two compression schemes form the backbone of MPEG, the actual implementation is immensely more sophisticated than described here. For example, a P-picture can be referenced to an I-picture that has been *shifted*, accounting for motion of objects in the image sequence. There are also bidirectional predictive-coded or **B-pictures**. These are referenced to both a previous and a future I-picture. This handles regions in the image that gradually change over many of frames. The individual frames can also be stored out-of-order in the compressed data to facilitate the proper sequencing of the I, P, and B-pictures. The addition of color and sound makes this all the more complicated.

В то время как эти две компрессионных схемы формируют основу MPEG, фактическое выполнение очень более сложно чем описанно здесь. Например, P-изображение может быть упомянуто к I-picture который был *сдвинут*, составляя(объясняя) движение объектов в последовательности изображения. Имеется также двунаправленное предективное кодированне или **B-pictures**. Они упомянуты к обоим и предыдущему и будущему I-picture.

Это обрабатывает области в изображении, которые постепенно переключают многие из структур(кадров). Индивидуальные структуры(кадры) могут также быть сохранены поврежденными в сжатых данных, чтобы облегчить надлежащее упорядочение I, P, and B-pictures. Добавление цвета и звука делает, это тем более усложненным.

The main distortion associated with MPEG occurs when large sections of the image change quickly. In effect, a burst of information is needed to keep up with the rapidly changing scenes. If the data rate is fixed, the viewer notices "blocky" patterns when changing from one scene to the next. This can be minimized in networks that transmit multiple video channels simultaneously, such as cable television. The sudden burst of information needed to support a rapidly changing scene in one video channel, is averaged with the modest requirements of the relatively static scenes in the other channels.

Основное искажение, связанное с MPEG происходит, когда большие разделы изображения изменяются быстро. В действительности, взрыв(импульс) информации необходим, чтобы не отставать от быстро изменяющихся сцен. Если скорость передачи данных установлена, средство просмотра обращает внимание на образцы "blocky"(блоки; кодовые группы) при изменении(замене) от одной сцены до следующей. Это может быть минимизировано в сетях, которые передают множественные каналы видео одновременно, типа кабельного телевидения. Внезапный взрыв(импульс) информации, необходимой поддерживать быстро изменяющуюся сцену в одном канале видео, усреднен со скромными требованиями относительно статических сцен в других каналах.