Lecture 2: Bits and Data representations

CAB203 Discrete Structures

Matthew McKague

Queensland University of Technology

matthew.mckague@qut.edu.au

Outline

Bit strings

Data representations

Text

Numbers

Other kinds of data

Readings

Next week

- Pace: 2.1 to 2.5

Outline

Bit strings

Data representations

Text

Numbers

Other kinds of data

Bits

All data in computers is stored in *bits*.

- A bit is something that has 2 *states*
- Usually we label the two states “0” and “1”

Bits are also a fundamental unit of *information*, and play an important role in information theory.

Examples of bits

Some examples:

- A mathematical variable x that can take values in $\{0, 1\}$
- A wire that has either low or high voltage
- A mechanical device with two possible positions
- Position of an on/off switch
- Heads or tails on a coin

A bit is a mathematical object used to model some real world property.

Uses of bits

Everything that happens in a computer involves bits:

- Inputting data
- Storing data
- Manipulating data (arithmetic, etc.)
- Outputting data

Data representations model data as combinations of multiple bits.

Bit strings

Bits are small. We usually use many bits together in *strings*. Examples:

- 0 (a single bit is also a bit string)
- 00000
- 10101110001
- 111111111111111111111111111111111111

Bit strings have a *length* which is just the number of symbols.

Bit string notation

Some notation:

- We'll use a bar over variables to indicate a string: $\overline{x}$
- The set of all strings of length n is $\{0,1\}^n$ (also called n-bit strings)
- *All* bit strings of all lengths are members of $\{0,1\}^*$
- The jth bit in $\overline{x}$ is $\overline{x}_j$ (j goes from 0 to $n-1$)
- For bit strings we most often count from the *right*, so $\overline{x}_0$ is the furthest right.

(We'll go over the $\{\cdot\}$ notation when we discuss Sets)

Number of bit strings

How many different bit strings of length n are there?

- Two choices for $\overline{x}_0$, 0 and 1
- *for each* choice of $\overline{x}_0$ there are two choices for $\overline{x}_1$, 4 total
- For each of the four choices for $\overline{x}_1\overline{x}_0$ there are again two choices for $\overline{x}_2$, total of 8.
- In general there will be 2^n possible bit strings of length n

Operators

An *operator* or *operation* is a mathematical object that transforms other objects. Examples:

- $+$ is a *binary* operator (transforms two objects). Eg. $1 + 2$ transforms 1 and 2 into 3.
- $-$ is actually two different operators!
  - As a binary operator, $-$ is subtraction, Eg. $2 - 1$ becomes 1
  - As a *unary* operator, $-$ is negation, Eg. -2 as a negative number.

Operators are extremely common in mathematical theories and are used in axioms to specify relationships between other objects. Eg. $a + 0 = a$.

Bit operations

Two types of bit operations:

- Operations on a single bit or pairs of bits
- Operations on bit strings

Operations on bits can also be thought of as logical operators, which we'll talk about later.

We'll use the notation common in programming languages like C and Python for bit operations.

NOT

NOT flips a bit: 0 becomes 1 and vice versa.

x	$\sim x$
0	1
1	0

this is often called a *bit flip*.

AND

AND is like multiplication.

x	y	$x \& y$
0	0	0
0	1	0
1	0	0
1	1	1

OR

OR is like addition, but what do you do with 2? Squash to 1.

x	y	$x \mid y$
0	0	0
0	1	1
1	0	1
1	1	1

XOR

XOR is another kind of addition, but squash 2 to 0, like parity.

x	y	x ^ y
0	0	0
0	1	1
1	0	1
1	1	0

$(\{0,1\}, \hat{}\,, \&)$ is a *Field*, equivalent to integers modulo 2.

Bit-wise operations

We can apply bit operations *bit-wise* to strings of the same length. If $\overline{z} = \overline{x} \,\&\, \overline{y}$ then

$$\overline{z}_j = \overline{x}_j \,\&\, \overline{y}_j$$

We just perform the operation on pairs of bits. Other operations work similarly.

$\{0,1\}^n$ is a group under bit-wise ^.

Bit shifts

We can move bits around in a string.

- Left shift by m bits drops the leftmost m bits and adds m 0's on the right:

$$011010 << 1 = 110100$$

$$100001 << 3 = 001000$$

- Right shift by m bits drops the rightmost m bits and adds m 0's on the left:

$$011011 >> 1 = 001101$$

$$100000 >> 3 = 000100$$

There are also *left rotations* where bits dropped on the left appear on the right, and analogously *right rotations*.

Concatenation

We can also *concatenate* bit strings, which joins them together. If $\overline{x}$ is an n-bit string and $\overline{y}$ is a m-bit string, then $\overline{z} = \overline{xy}$ is a $(n + m)$-bit string.
Example: $\overline{x} = 000$ and $\overline{y} = 11$ then $\overline{xy} = 00011$.

The set of all bit strings $\{0, 1\}^*$ forms a *monoid* under concatenation.

Bit manipulation

Most CPUs work on 8, 32, or 64 bits at a time, not individual bits. So we can't manipulate a single bit directly, but sometimes we want to. But we can do this using bitwise operators.

Some things we might want to do:

- Turn some bits on (set them to 1) aka *set* the bit
- Turn some bits off (set them to 0) aka *clear* the bit
- Flip some bits (NOT them) aka *toggle* the bit
- Test if a bit is 0

This is extremely common in systems programming (operating systems, networking, etc.) and in embedded systems (microcontrollers).

Bitmasks

We use the concept of a *bitmask* (or just mask) to manipulate groups of bits. Assume 4 bit strings with $\overline{x} = 1100$.

- Mask for bits 0 and 2: $\overline{m} = 0101$
- Turn on bits 0 and 2:
$$\overline{x} \,|\, \overline{m} = 1101$$
- Turn off bits 0 and 2:
$$\overline{x} \,\&\, (\sim\overline{m}) = 1000$$
- Turn off all bits except 0 and 2:
$$\overline{x} \,\&\, \overline{m} = 0100$$
- Flip bits 0 and 2:
$$\overline{x} \,\hat{}\, \overline{m} = 1001$$

Bit strings and masks in Python

```
>>> 0b101                      # 0b prefix for binary constants
5
>>> bin(0b101)                 # bin() gives binary representation
'0b101'                        # comes out as a string
>>> print(bin(5))
0b101
>>> print(0b101)               # printed as integer by default
5
>>> bin(0b1100 & 0b1010)       # AND
'0b1000'
>>> bin(0b1100 | 0b1010)       # OR
'0b1110'
>>> bin(0b1100 ^ 0b1010)       # XOR
'0b110'
>>> bin(~0b10)                 # NOT all bits
'-0b11'                        # This is a negative number,
                               # explained next week
```

Bit strings and masks in Python

```
>>> ctrlmask = 1 << 3           # create a mask for bit 3
>>> x = 0
>>> x = x ^ ctrlmask            # flip bit 3
>>> bin(x)
'0b1000'
>>> x = x ^ ctrlmask            # flip bit 3 again
>>> bin(x)
'0b0'
>>> x = x & ~ctrlmask           # turn off bit 3
>>> bin(x)
'0b0'
>>> x = x | ctrlmask            # turn on bit 3
>>> bin(x)
'0b1000'
>>> ctrl = (x & ctrlmask) != 0  # test bit 3 (on = true)
>>> ctrl
True
```

Outline

Bit strings

Data representations

Text

Numbers

Other kinds of data

Representing data as bits

Some types of data representations that we will need:

- Characters
- Strings of characters (text)
- Integers
- Decimal numbers (floats)

In practice, most other types of data are converted to numbers or strings or some larger structure with numbers and text inside.

An example of data that are not encoded via characters or numbers are *CPU instructions*. Their encoding is defined by how the computer interprets them.

Outline

Bit strings

Data representations

Text

Numbers

Other kinds of data

Characters as bits

Characters are just symbols. Some types:

- Symbols from writing systems (e.g. Latin letters, Devanagari letters, symbols from Kanji)
- Numerals (0, 1, 2, . . . 9)
- Diacritics (accents)
- Punctuation (. , ; : ”)
- Mathematical symbols ($+$, $\times$, $-$, %)
- Braille symbols
- Unprintable characters (space, newline, tab)

Encodings

To represent characters as bit strings we need an *encoding*. We need:

- A set of characters to represent (e.g. the Latin alphabet)
- A length *n* for our bit strings
- A *mapping* from characters to $\{0,1\}^n$

A mapping needs to have *exactly one* bit string for each character, and no two characters sharing the same bit string.

Example encoding

Let's encode the set of characters { a, f, z, 7 } in 2-bit strings

String	Character
00	z
01	a
10	7
11	f

This isn't a very good encoding!

- The set of characters isn't useful
- There is no logical ordering
- Different types of characters (letters, numbers) are mixed together

Lexicographic ordering

A common way of ordering bit strings is *lexicographic ordering*. This is like alphabetical ordering:

- 0 comes before 1
- Compare strings one bit a time from left to right
- At the first bit where the strings differ, the one with a 0 goes first
- If one string is longer, then pad the shorter on the right with empty spaces (empty spaces come before 0)

Examples:

- 000 comes before 100 (like bat comes before cat)
- 000 comes before 001 (like are comes before art)
- 011 comes before 100 (like bzz comes before caa)
- 01 comes before 010 (like at comes before ate)

ASCII

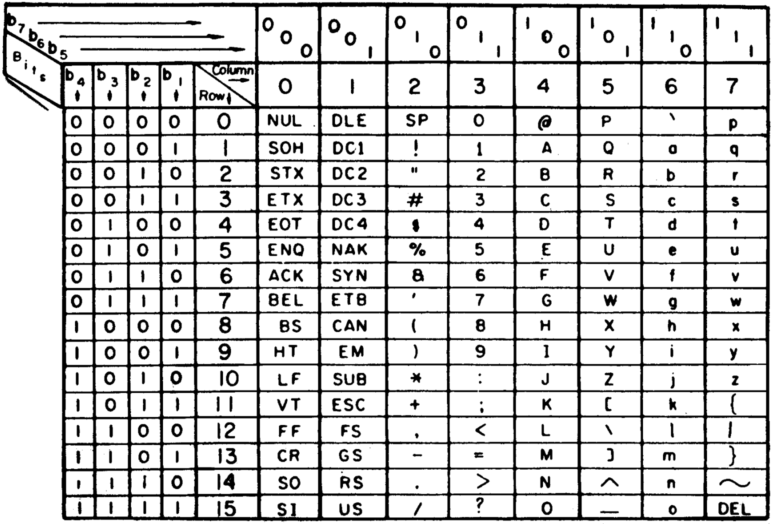
ASCII is an older encoding for English text. It was designed for teleprinters before computers were widespread:

- 7 bit strings
- 128 characters
- Upper, lower case Latin characters, numbers, punctuation, math symbols, space, newline, etc.
- Special characters like BEL which represent a bell ringing on the printer, ESC, NUL that had special meanings for the printer
- Hundreds of different extensions to 8 bits, eg. ISO 8859

The Baudot code used 5-bits and was used for telegraphs. Several 6-bit codes were used on early mainframes.

ASCII chart

USASCII code chart



Bits b7 b6 b5 →					0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b4	b3	b2	b1	Row ↓ Column →	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	-	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

Features of ASCII

- Letters ordered lexicographically
- Numbers 0–9 ordered lexicographically
- Blocks of related characters (upper case, lower case, numbers, etc.)
- Upper vs. lower case is just one bit
- Usually we store 8 bits, with the leftmost bit set to 0

UNICODE

Unicode is a modern system of character encodings that supports most writing systems.

- About 137 000 characters supported
- Support for most modern and some historic writing systems
- Mathematical symbols, punctuation, emoji, etc.
- Multiple encodings for a common set of characters

The UNICODE list of official emoji speficies 3633 characters, including “pile of poo”, “alien monster”, and “woman facepalming: medium skin tone”

UNICODE encodings

Unicode assigns a *code point* (a hexidecimal string) for each character. There are several different encodings from code points to bit strings

- UTF32 uses 32 bits for each character, encoding code points directly
- UTF16 uses one or two 16-bit strings per code point, making it a *variable length encoding*
- UTF8 uses between one and four 8-bit stings per code point, and is hence also a variable length encoding. It is backwards compatible with ASCII for the original 7-bit ASCII character set

UTF8 is the most common encoding. Python strings are UTF-8 encoded by default.

Character strings

We usually need more than one character at a time. So we have *strings*.

- Start with a sequence of characters
- For each character, find its representation as bits
- Push the individual representations together to make one long string of bits

String subtleties

To interpret a string of bits as characters we need to know

- How long each character representation is
- When to stop

For fixed length encodings like UTF32 and ASCII we just need to know when to stop. For variable length encodings like UTF8, we need to have a way of recognising where a particular character stops.

C and Pascal strings

Every computer language has its own way of storing strings

- Pascal (and most modern languages) stores the number of characters along with a string
- C uses *null-terminated strings*: After the last character it stores bits 00000000 (ASCII symbol NUL) to signal the end.

The standard C libraries are not *8-bit clean*. You can't store the NUL character as part of a string. More modern libraries fix this, but you need to keep track of the length separately.

C-style example

Let's encode "the" as a C-style string.

- "t" has ASCII representation 01110100
- "h" has ASCII representation 01101000
- "e" has ASCII representation 01100101

Add a NUL on the end, we get
01110100 01101000 01100101 00000000

Outline

Bit strings

Data representations

Text

Numbers

Other kinds of data

Integers

How to store numbers? Some properties we would like

- Lexicographical ordering of bit strings is same as ordering numbers
- Easy to do math on encoded numbers

Base-10 notation

The usual way of writing numbers in English

- *Numerals* are 0,1,2,3,4,5,6,7,8,9
- Numbers encoded as a string of numerals
- Position starts at 0 at the right-most numeral (like bit strings)
- Position j gets a multiplier of 10^j
- Add up all the values

Example: 2021
Starting from position 0 (rightmost)

- 1 has a multiplier of $10^0 = 1$
- 2 has a multiplier of $10^1 = 10$
- 0 has a multiplier of $10^2 = 100$
- 2 has a multiplier of $10^3 = 1000$
- Total is 2021

Binary representation

Binary numbers are analogous to base-10 notation:

- Numerals are 0,1 (i.e. bits)
- Numbers encoded as string of numerals (i.e. a bit string)
- Position starts at 0 at the right-most numeral (like bit strings)
- Position j gets a multiplier of 2^j
- Add up all the values

Example: 1010
Starting from position 0 (rightmost)

- 0 has a multiplier of $2^0 = 1$
- 1 has a multiplier of $2^1 = 2$
- 0 has a multiplier of $2^2 = 4$
- 1 has a multiplier of $2^3 = 8$
- Total is 10

4-bit binary

Remember these! Or at least get quick at working them out.

base-10	base-2	base-10	base-2
0	0000	8	1000
1	0001	9	1001
2	0010	10	1010
3	0011	11	1011
4	0100	12	1100
5	0101	13	1101
6	0110	14	1110
7	0111	15	1111

4 bits is called a *nibble*. The Intel 4004 was the first commercial microprocessor (on a single chip) and was a 4-bit CPU.

Binary Representation

“There are 10 types of people in this world. Those who understand binary and those who don’t.” - Ian Stewart, 1995

8-bit binary numbers

Let's store positive numbers $0 \ldots 255$ as 8-bit strings.

$$\sum_{j=0}^{7} 2^j \overline{x}_j$$

Example: What number is 01000010?

$$0 \cdot 128 + 1 \cdot 64 + \cdots + 1 \cdot 2 + 0 \cdot 1 = 66$$

Adding binary numbers

We'll start by adding two 1-bit numbers:

$$\begin{aligned} 0+0 &= 0 \\ 1+0 &= 1 \\ 0+1 &= 1 \\ 1+1 &= 10 \end{aligned}$$

In the last line we needed to *carry* because we had a 2.

Adding as bit operations

Suppose we have 1-bit binary numbers $\overline{x}$ and $\overline{y}$ and we want to calculate the sum in binary representation. Then the sum is a 2-bit string $\overline{z}$ where

$$\begin{aligned} \overline{z}_0 &= \overline{x}_0 \,\hat{}\, \overline{y}_0 \\ \overline{z}_1 &= \overline{x}_0 \,\&\, \overline{y}_0 \end{aligned}$$

Adding as bit operations

More generally, for n-bit binary numbers $\overline{x}$ and $\overline{y}$, the sum in binary $\overline{z}$ is a $n+1$-bit binary number where

$$\begin{array}{rcl} \overline{z}_j & = & \overline{x}_j \mathbin{\hat{}} \overline{y}_j \mathbin{\hat{}} \overline{c}_j \\ \overline{c}_{j+1} & = & (\overline{x}_j \,\&\, \overline{y}_j) \,|\, (\overline{x}_j \,\&\, \overline{c}_j) \,|\, (\overline{y}_j \,\&\, \overline{c}_j) \end{array}$$

The string $\overline{c}$ is the carry bits (take $\overline{c}_0$ to be 0.) The equation for $\overline{c}_{j+1}$ just says that it is 1 when $\overline{x}_j + \overline{y}_j + \overline{c}_j$ is 2 or 3.

Modular arithmetic in binary

In a typical CPU, integers have a fixed number of bits (usually 8, 32, or 64). What happens when we have a number that is too big?

Assume 8-bit numbers. Add 10000000 (128) and 10000001 (129) to get 100000001 (257, 9-bits). We drop the leftmost bit to get 8-bits: 00000001 (1).

This is exactly the same as modding out by $2^8 = 256$.

$$257 \mod 256 = 1$$

So CPUs usually implement integer arithmetic modulo 2^n where n is the number of bits.

Negative numbers

There are several ways of encoding negative numbers. We'll use *2's complement* encoding. Eg. assuming 8 bit integers:

- What is $-1 \mod 256$? 255
- -1 is indistinguishable from 255 in arithmetic modulo 256
- So represent -1 with 255 in binary
- Note: we can't represent 255 any more!

More generally, represent $-x$ with $256 - x$ in binary.

There are other ways of representing negative numbers. Some others are signed magnitude, 1's complement, and excess notation.

More negative numbers

Properties of 2's complement:

- For n bits, we can represent -2^{n-1} through to $2^{n-1} - 1$
- The leftmost bit is 0 for positive numbers, 1 for negative numbers
- Addition is modulo 2^n
- The positive numbers 0 to $2^{n-1} - 1$ haven't changed

The difference between 2's complement encoding and regular binary encoding is how we interpret the bit strings. Arithmetic works the same, so we don't need special CPU instructions for subtraction.

3-bit 2's complement

bit string	2's complement interpretation	binary interpretation
000	0	0
001	1	1
010	2	2
011	3	3
100	-4	4
101	-3	5
110	-2	6
111	-1	7

2's complement corresponds to *signed integers*, while binary corresponds to *unsigned integers* in most programming languages.

Hexadecimal

You will often see *hexadecimal* numbers in computer science. You can think of these in two ways:

- As a Base-16 number system: the numeral in position j gets a multiplier of 16^j.
- As a compact way of writing bit strings

Most of the time we use it just for writing bit strings.

Hex

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

Interpreting HEX

Write 4F as an 8-bit string:

$4 \to 0100$, and $F \to 1111$. So we get

$$01001111$$

Write 4F as a base-10 number:

4 is just 4, and gets a multiplier of 16^1 because it is in position 1.
F is 15, and gets a multiplier of $16^0 = 1$ since it is in position 0.
The total is:

$$4 \cdot 16 + 15 = 79$$

Why do we use hex?

Hex has a few advantages over bit strings or base-10 for *humans* in many applications:

- Shorter than bit strings
- One hex numeral is always exactly 4 bits, so bit strings divide up nicely
- For, 8 bits, we always need exactly 2 numerals (could be 3 for decimal)
- Relatively easy to work out individual bits by hand

Compare:
11000000.10101000.00000000.00000001
192.168.0.1
C0:A8:00:01

Scientific notation

Integers are not always enough. We might need to encode:

- Very small numbers (e.g. size of an atom)
- Very large numbers (e.g. size of the universe)

The usual way to do this is with *scientific notation*:

- Size of a hydrogen atom: 5.0×10^{-11} meters
- Size of the observable universe: 8.8×10^{26} meters

Scientific notation also implies a level of *precision*. The number is not known exactly.

Scientific notation

Parts of scientific notation:

$$\pm a.bc \times 10^e$$

- $\pm$ is the *sign*
- $a.bc$ are the *significant digits*
- e is the *exponent*
- 10 is the *base*

The base always matches the base that the significant digits and exponent are written in. In this case, everything is in base 10.

Scientific notation, base 2

We can consider scientific notation is base 2 as well.

- Significant digits are bits
- Base is 2
- Exponent is written in binary

Example:

$$1.1010 \times 2^{10}$$

Floating point numbers

The computer version of scientific notation is *floating point numbers*. We encode (in binary):

- The significant digits
- The exponent
- The sign (positive or negative)
- No need to encode the base (it is understood to be 2)

Example: to encode 1.1010×2^{10} we need to store:

- This is a positive number
- The significant digits are 11010
- The exponent is 10

IEEE half-precision

The standard for floating point numbers is IEEE 754. For 16 bits it looks like

$$\overbrace{0}^{s}\ \overbrace{10101}^{e}\ \overbrace{1010101010}^{f}$$

- s (1-bit) encodes the sign
- e (5 bits) encodes the exponent
- f (10 bits) encodes the significant digits (drop the leading 1)

IEEE 754 uses excess notation for encoding positive and negative exponents, rather than 2's complement.

Python numbers

Python has two basic number types:

- `int`: arbitrary length integers. Defaults to 32-bit 2's complement, but internally allocates more memory as needed.
- `float`: IEEE 754 Double precision: 64-bit floating point

Python numbers

```
>>> # really big integers are just ints
>>> type(1000000000000000000000000000000000000000000000000)
<class 'int'>
>>> type(10)     # same type for all lengths of integer
<class 'int'>
>>> type(1.0)    # decimals stored as floats
<class 'float'>
>>> 9 / 2        # regular division always returns a float
4.5
>>> 9 // 2       # floor division returns an integer
4
>>> type(9 / 2)
<class 'float'>
>>> type(9 // 2)
<class 'int'>
>>>
```

Python hex

```
>>> "{:x}".format(65532)  # format integer in hex
'fffc'
>>> 0xfffc                # hex literals
65532
>>>
```

Outline

Bit strings

Data representations

Text

Numbers

Other kinds of data

Other kinds of data

Besides text and numbers, what else might we want to represent?

- Pictures
- Videos
- Sounds
- Vectors (eg. x-y coordinates on screen)
- Various sciency things (temperature, salinity, etc.)
- . . .

How to represent other kinds of data?

For most kinds of data, we combine text and numbers:

- Metadata: description of how the data is encoded:
    - Picture size in pixels
    - Audio sampling rate
    - Colour or sample bit depth
    - Length of data
    - ...
- The data, usually as a string of numbers, in some well defined order

Often we need nested structures, for example a string of pixels, where each pixel is three numbers.

Non-numerical data

Some data is neither characters nor numbers and have special interpretations:

- Individual bits (eg. CTRL down, or CTRL up)
- IP or MAC addresses
- CPU instructions
- Protocol status (eg. TCP SYN and ACK bits)

These are usually just stored as bit fields with known lengths/locations. Eg. TCP SYN bit is always bit 1 of byte 13 of the packet header.