8-bit Workbench™ ED

Operation Manual

8BitFlux.com

Get hands-on with number systems and explore the computer byte. Toggle eight bits and see their *decimal*, *hexadecimal* and *octal* value. Annotations at bit-level break down the displayed value and make comprehension easy.

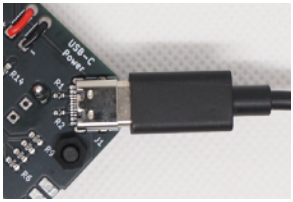
Includes a built-in game: beat the high score by toggling in the correct bits for each number system. A great **s**cience, **t**echnology, **e**ngineering, and **m**athematics (STEM) learning kit.

This manual shows how to operate the 8-bit Workbench™ ED and gives some lessons that teach how computers work with numbers, with a vintage twist. For details on how to assemble the kit, *see* the separate **Assembly Instructions**.

Important: **do not touch** the microprocessor (the black rectangular box marked with 'U1') as electrostatic discharges can damage it.

Supplying Power

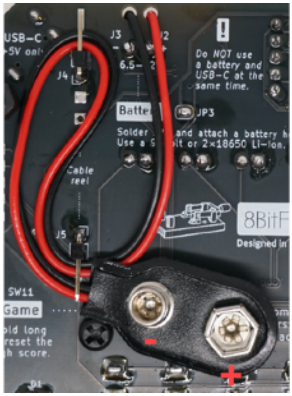
The device can be powered by **USB-C** or use the **battery** connectors.



A connected USB-C cable.

Plug in a USB-C cable to the power connector J1 (*see* figure). Attach the other end to a USB charger, laptop or power bank. It needs +5V only. Please note that the connector does not support the transfer of data.

Attach a battery to the wire connectors J2 and J3. This input is versatile and accepts a voltage range from 6.5V to 12V. For example, a 9-volt block or two 18650 Li-Ion batteries can be used. A clip is supplied with the kit to use a 9-volt. *See* step 6 of the Assembly Instructions.



9-volt battery clip stowed away on the cable reel.

Important: make sure the **polarity** is correct and **not** reversed when attaching the 9-volt block battery. *See* figure, always stow away the battery clip in such a way that the shiny connectors do not point to the board, as this can cause a short circuit which damages the board. Applying a plastic spray (like *Plastik 70*) can prevent this.

Power-On with Self-Test

Once power is connected, slide the power switch to the **ON**-position to turn on the device. A power-on **self-test** runs.

The test emits the three green LEDs (next to *Decimal*, *Hex* and *Octal*) in sequence moving from top to bottom. At the same time the three digits emit an '8' moving from right to left. After the self-test, it goes in *Decimal* mode and the digits show the current decimal value of the eight bits that are toggled in.

Note that a battery can stay connected to the board after the device is turned OFF, the design prevents it from draining.

Explore using Guided Lessons

Learn by doing and explore the functionality of the 8-bit Workbench™ ED, with these lessons. Beginners can follow the lessons in sequence, others can just skip to the lessons that attract their interest. Be prepared for some history of computing as well.

Have the Workbench ready on your desk and attach power. Make sure all toggle switches point down and the *Flip the Bits*-switch is set to *Normal* (up-position).

For teachers, feel free to use (or adapt) these lessons to kickstart a new course. Multiple boards can be used in a classroom setting, the lessons can be a student's guide. All lessons are **licensed** under the *CC BY-SA*, for details *see* https://creativecommons.org/licenses/by-sa/4.0/

0: Number Systems

About 40,000 years ago (far before the year zero), stone age people documented their **counting** using *tally sticks*. For example, a piece of animal bone was carved with markings to represent the amount of cattle a herder owned. Hence the **tally marks**, an early number system, that we still know today.

Counting with tally marks from 1 to 5 (one hand).

It is not a coincidence that this way of counting maps to a hand with five fingers. Easy for counting by humans, but do computers have hands as well?

Another number system that we are all familiar with is **decimal**. Compared to the tally marks this uses a *positional* notation instead of a *symbol* notation. Meaning

© 2026, 8BitFlux.com

that the position of a single symbol (like a '2' or '3') affect its value (30 is far off 3). A single 'symbol', in one position, is called a **digit**.

Look at the *mechanical odometer* of my motorcycle. It has six of these digits and keeps track of the total amount of kilometers the bike has run: `98239` at the moment.

Mechanical odometer of a motorcycle.

When driving the bike (and the meter being mechanical) imagine seeing the right-hand digit turning slowly from `9` back to `0` again. Every time it does that, it takes the higher digit (the `3`) to the left with it to make a `4`. Then it keeps going from `0` to `1` and `2` again. Etcetera.

Lets take this mechanical counter apart in our heads. Each digit is a round dial with the symbols printed on the side: `0` to `9`. Including the zero this is a total of 10 unique symbols.

This '10' is called the **base** (or **radix**) of a number system. Decimal is a **base-10** number system. Each digit to the left is worth 10 times as much as the digit to the right of it. A base can also be seen as the amount of 'letters' in the 'alphabet' of the number system.

Turn on the Workbench. The **green** LED indicates the active number system, press the **Mode** button to change this. How many digits does the Workbench have? Which number systems can it display? What are their *bases*?

`Dec`
`Hex`
`Oct`

Discover that the more switches you move upwards, the more digits are 'being used' in the left-going direction. This all has to do with the nature of the the three number systems: they all use a *positional notation*, like the odometer of my motorcycle.

`Hex`
`Oct`

Use the Mode button and toggle switches randomly to figure out the 'alphabet' of the two number systems that are used specifically in computing. What are their possible unique 'symbols'? How many should there be for each one? Do both use all three digits?

The theory of number systems is important to understand how computers count the way they do. Number systems set the **rules for counting**. *Decimal*'s rules are different than *hexadecimal*'s or *octal*'s. But they have similarities, like they all use the positional notation and use digits to show their values.

© 2026, 8BitFlux.com

1: Binary, Bits and Bytes

The **binary** number system was invented by the mathematician *Thomas Harriot* around the turn of the 17th century, long before computers even existed. He used it in experiments determining the weight of wine. Binary is a **base-2** number system, meaning it has just two symbols: `0` and `1`.

Portrait claiming to be Thomas Harriot (1602).

Imagine rebuilding the odometer of the previous lesson: let it have a `0` and a `1` only for each digit (or dial). When a digit reaches `1` it turns over to `0` and the digit to the left of it turns over too (if this digit is `1` as well, it also turns over to `0` and takes the next with it, and so it goes on). In binary, each digit to the left is worth 2 times as much as the digit to the right of it (it doubles value).

Look at the back of the Workbench: a legend lists the binary numbers counting from 0 (`0000`) to 15 (`1111`). Observe that number systems with larger bases pack more values into each single position or digit.

Engineers ended up adopting the binary system for electrical computers because it matches the behavior of its components. A switch, relay, vacuum tube or **transistor** all have *two states*: they can be OFF (`0`) or ON (`1`), having a low or high voltage level present. Each electrical 'switch' maps to a ***bi**nary digi**t***, or a **bit**.

The Workbench has 3 'big' transistors right next to the digits, marked Q1-Q3, they are used for the digit display. Another 20,000 tiny transistors are inside its microprocessor (the small black box marked U1). They do all the 'hard' work.

Dec

Turn on the device. The *Binary Keyboard,* with 8 toggle switches, can be used to input 8 bits. Toggle each switch individually, going from right to left and see that the number (in decimal) doubles each time when doing so (1, 2, 4 etc.). Notice that the *annotations* at the *Decimal*-row below the LEDs match these numbers.

Dec

Experiment with numbers in between by toggling up more than one switch at a time. Look at the annotations, how are the digits being calculated?

When pressing a key on a keyboard, binary data that represents this key is being send to the computer. For example, an '`A`' is `0100 0001` in **ASCII**. Touching the screen on your phone triggers binary data containing your finger position being send to the operating system running on your phone.

© 2026, 8BitFlux.com

Binary **data** is *everywhere,* a very long list of only `0`'s and `1`'s.

By modern standards, a size of 8 bits is called a **byte** (or **octet**). The bits within the byte can be indexed 0 to 7, going right to left. The bit with the lowest order place (bit 0) is called the *least significant bit* (LSb). The bit on the far left (bit 7) is the *most significant bit* (MSb).

In early computing, a decimal number was stored in half a byte (4 bits) and was called *binary-coded decimal* (**BCD**). This way, decimals up to 99 could be stored in one full byte, like `0101 0011` for number 53. In BCD, those half bytes can be further *packed* to form higher decimal numbers. It is important that the computer knows it is dealing with packed-BCD data when it is fed with `0101 0011`, otherwise it will interpret it as 'something else' or an '`S`'.

Try out BCD on the Workbench. Toggle the switches and try to count from 0 to 9. Use the *four* least significant bits for this.

Dec

Toggle in the packed-BCD byte `0101 0011`, does it show 53? Why (not)?

Hex

Set the Mode to *Hex*. Do the counting again from 0 to 9. What do you notice? Toggle in the BCD byte for 53. Is it correctly displayed by the digits now? Why (not)?

Again look at the legend at the back of the Workbench, all BCD numbers are marked with the vertical line.

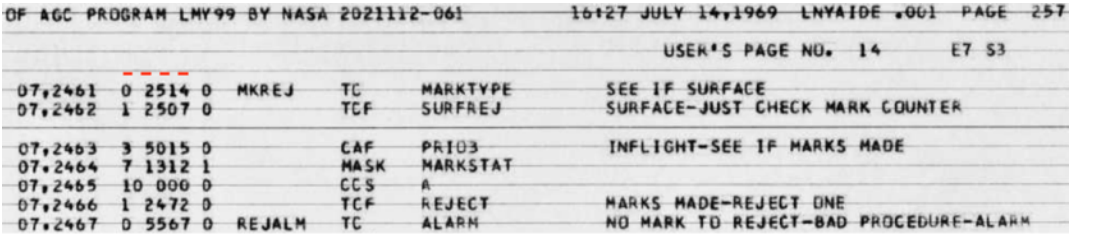
2: A word on Words

So how do we tell computers what to do? By feeding them, again, binary *data*. However these are then called **instructions**, which are executed by the computer's processor.

Imagine a processor taking 'bites' (not bytes) of a stream of zero's and ones. It keeps 'eating' and executing. Depending on the kind of processor (and the design of the computer) these 'bites' can be big or small. They are several bits long. One bite is called a **word** and it is of a certain *size* (amount of bits).

Early day computers typically had word sizes that were **divisible by 3**. The *PDP-8* made by *Digital Equipment Corporation (DEC)* is an example. Released in 1965, it was a 12-bit computer, meaning it had a word size of 12 bits. Early mainframes (from the 1950s to early 70s) often used 36-bit words, among them were *IBM's* and *UNIVAC's*. For these word sizes it was convenient to use **octal**, because an octal digit maps to 3 bits. That is, a 6-bit word can be represented by 2 octal digits.

Let's take a look at the computer that landed men on the moon: the *Apollo Guidance Computer (AGC)*. A very small part of the source code that went to the moon is shown below.



```
OF AGC PROGRAM LMY99 BY NASA 2021112-061          16:27 JULY 14,1969  LMYAIDE .001  PAGE  257

                                                  USER'S PAGE NO.  14       E7 S3

                 ----
07,2461  0 2514 0  MKREJ    TC      MARKTYPE        SEE IF SURFACE
07,2462  1 2507 0           TCF     SURFREJ         SURFACE-JUST CHECK MARK COUNTER

07,2463  3 5015 0           CAF     PRIO3           INFLIGHT-SEE IF MARKS MADE
07,2464  7 1312 1           MASK    MARKSTAT
07,2465  10 000 0           CCS     A
07,2466  1 2472 0           TCF     REJECT          MARKS MADE-REJECT ONE
07,2467  0 5567 0  REJALM    TC      ALARM           NO MARK TO REJECT-BAD PROCEDURE-ALARM
```

Assembly source listing of the Apollo Guidance Computer (AGC) from 1969.

The column with the *dashed line* above it contains the actual **instruction-bits** that were fed to the processor. This being not one chip, but actually thousands of the world's first ICs were used in the AGC.

Look at Apollo's source listing, each line representing one instruction, a 'step' in the program. How many *digits* are used here to represent the actual bits of an instruction?

? While the AGC flies to the moon, it 'eats' the bits `0000101010011100` at the first instruction. What *number system* is used in this source listing? Find out using the Workbench. Tip: do not use the two *most significant* toggle switches.

Rewrite the bits of the 'eaten' word in such a way that it is easier to read and that it becomes clear what number system is being used. What is the word size of the AGC?

An AGC-instruction (*dashed line*) consists of an *operation* (like add, subtract, etc.) and a memory *address*, separated by a space. Both correspond to the second and third columns that are 'human readable'. For example: `0` means operation `TC` and `2514` means address `MARKTYPE` in the first line.

? At a certain moment the AGC 'eats' `000101101110111`. To what memory address (a number) is this instruction referring to? What is it referred to by humans?

? What is this same address in the other number system used in computing? How many digits are being used now?

© 2026, 8BitFlux.com

3: Nibbles, not Nabbles

Is this the third or the fourth lesson? Good, you are learning.

We have completed half a byte of lessons after this one, it is called a **nibble**, being 4 bits. A *binary-coded decimal* (BCD) is stored in a nibble. A **hexadecimal** digit is represented by a nibble.

See the legend at the back, all possible nibbles are in the column at the right. Observe that all binary values match one symbol in *Hex*, that goes from `0` to `F`. How many ***unique*** **values** can a nibble contain?

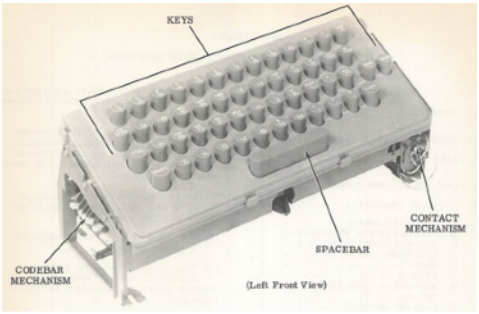
High and *low nibbles* are marked by labels: below the digits and on the Binary Keyboard. Added to this, nibbles are separated by *solid* vertical lines. The dashed ones indicate the octal bit groups.

Hex In *Hex* mode, toggle in the **maximum value** of the low nibble. What do the digits display? Notice that maximum values are shown in italics in the annotations below the LEDs.

Dec What is the maximum value of a nibble in decimal? Is this the same as the amount of *unique* values? Why (not)? How many digits are being used now?

Hex Back to *Hex* again. Use the ‘Flip the Bits’-switch to, guess what, flip the bits. What nibble is showing its maximum value now? Observe that the digits are also being ‘flipped’. Why is the left digit always off?

Dec What happens when the same is done in decimal? How many unique values can a full byte contain? What number system would you think is more suitable to use in computing? Why?



Keyboard of the Model 33 teleprinter, released in 1963 by Teletype Corporation. One of the first products to support ASCII.

During the 60s, the evolution of **ASCII** (a standard for representing characters in binary) was a turning point in computing with regard to the word sizes used.

Before ASCII, computers fitted characters into 6 bits most of the time and supported upper case only. Word sizes were mostly *divisible by 3* and octal was a clear-cut way to represent their data.

See the previous lesson that goes more into this.

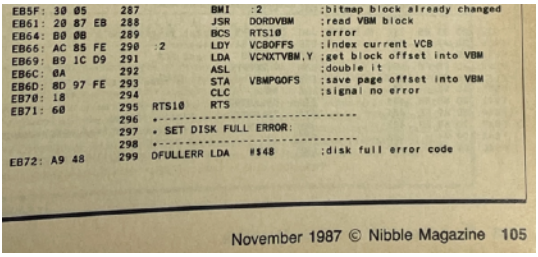
© 2026, 8BitFlux.com

An original *ASCII character* is 7 bits long (or 8 with a *parity* bit), has support for lower case and fits nicely in a byte. ASCII became more popular and computers started using word sizes with a **multiple of 4** (like 4, 8, 16, 32, 64).

Throughout the 1970s, this was further amplified by the appearance of popular 8-bit micro-processors, like the *Intel 8080*, *MOS Technology 6502* and the *Zilog Z80*. By the end of the 70s, beginning 80s, ***personal* computing** at everybody's home really took off.

As a result, the use of octal faded away and **hexadecimal** became the de facto standard.

A lot of computer programs and games were shared among people. There was no Internet, as we know today, so they were published in magazines (*see* figure). Remember the 90s configuring a Sound Blaster card in MS-DOS? The default address of `220` was a hex value.



```
EB5F: 30 05     287          BMI  :2        ;bitmap block already changed
EB61: 20 87 EB  288          JSR  DORDVBM    ;read VBM block
EB64: B0 08     289          BCS  RTS10      ;error
EB66: AC 85 FE  290 :2       LDY  VCBOFFS    ;index current VCB
EB69: B9 1C D9  291          LDA  VCNXTVBM,Y ;get block offset into VBM
EB6C: 0A        292          ASL             ;double it
EB6D: 8D 97 FE  293          STA  VBMPGOFS   ;save page offset into VBM
EB70: 18        294          CLC             ;signal no error
EB71: 60        295 RTS10    RTS
                296 ;-----------------------------
                297 ; SET DISK FULL ERROR:
                298 ;-----------------------------
EB72: A9 48     299 DFULLERR LDA  #$48       ;disk full error code
```

November 1987 © Nibble Magazine 105

Age-yellowed page out of Nibble Magazine (1987). Computer programs were shared among readers using hexadecimal source listings.

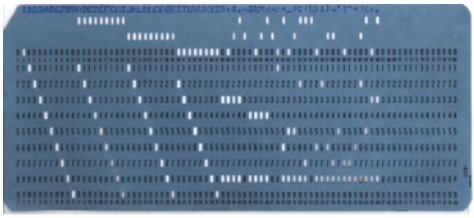
Computers today have become more 'hungry', most of them use a word size of 64 bits (this even includes *Nvidia's Blackwell* chip used for AI). Just imagine all those 64 toggle switches… 16 digit displays… but still conveniently written down in 8 pairs of hexadecimal nibbles: `50 75 7A 7A 6C 65 72 21`.

4: Meaningful Codes

Di-di-dit, *dah-dah-dah*, *di-di-dit*… the sounds that were heard at a naval base in 1872. Meaning that a boat was in distress and called out 'SOS' in *Morse code*. An early example of (character) **encoding**.

Encoding is changing information from one format to another. Morse code converts letters of our alphabet to *short* and *long* sounds (or flashes of light). The other way around is called *decoding*.

Computers 'talk' binary with zeros and ones only. As humans, we like to communicate different. Meaning a lot of *encoding* and *decoding* is needed in computing.



Punched card containing EBCDIC data, an 8-bit encoding used by IBM in the 60s.

© 2026, 8BitFlux.com

Some examples; a computer treats `0101 0011` as being 53 when packed-**BCD** encoding is used (*see* lesson 1). A binary `0100 0001` results in an '`A`' when the *computer program* interprets its data as being **ASCII**-encoded.

Watching a movie (a stream of video data) on your favorite streaming service involves the modern *H.264* or *AV1* encoding.

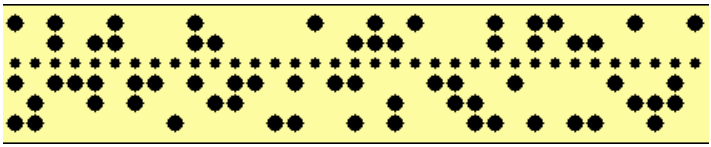
Paper-tape, an early form of data storage. Same concept was already used to automate looms in the 18th century.

A processor cannot take apart binary data from instructions, it just assumes the latter and processes this in sequence, being a machine after all. It is the **computer program** itself (the list of instructions) that determines:

- *what* pieces of the *data* have to be treated as such,
- *where* they are (e.g. memory address, paper tape input device, on disk)
- and *how* they should be treated, i.e. what meaning has the data, agreed upon by **encoding**.

Hex

We received a secret message on paper-tape. Fortunately we know it is **Baudot** encoded. The single row with the tinier holes in the paper are used to transport the tape, they represent no data and sit between bit number 1 and 2. What does the message say? Decode it using the Workbench and the table below.



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x	*NUL*	E	*LF*	A	*SP*	S	I	U	*CR*	D	R	J	N	F	C	K
1x	T	Z	L	W	H	Y	P	Q	O	B	G	*FIGS*	M	X	V	*LTRS*

Letter set of the 5-bit Baudot–Murray code (expressed here as hexadecimal numbers). It was used in telegraphy and as teletype code, introduced in 1932, before ASCII. 'SP' means space.

In fact, a *number system* by itself is also a form of foundational encoding. It transforms mathematical values into a readable and structured format, defined by its own rules.

© 2026, 8BitFlux.com

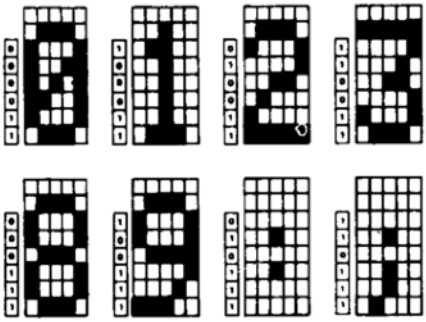
5: Typing on a Television

It was like *magic*: you pressed a key and the character appeared on your television set. People were used to *typewriters* in their living room, with letters produced on a piece of paper only. This was next-level, it was called a **TV Typewriter**.

Designed by *Don Lancaster*, it appeared in Radio-Electronics magazine back in 1973. People handy enough with electronics were able to build it. Later on they appeared in kit-form (*see* figure).

Example of an early TV Typewriter, the SWTPC CT-1024 Terminal. Sold as a kit in 1974.

At that time, **ASCII** got some foot on the ground and the standard was being used increasingly in electronic parts. For example, an *ASCII keyboard* or an *ASCII character generator chip* (*see* figure) were parts that a hobbyist could buy at an electronics store. Both were used by Don to design his TV Typewriter.



Examples of the characters (made of pixels) stored in the 2513 character generator IC that used ASCII.

So it had a keyboard, a screen, it used memory, had a cursor and some even had a serial interface. So was it a **personal computer**? No, it was not. It lacked a *microprocessor* and *a way to store programs*, all in a single casing.

The ASCII standard uses *printable characters* and *control codes*, e.g. to let the computer do a carriage return (`CR`) or a delete (`DEL`). *See* the chart on the next page, observe the bits matching with the same symbol in the character generator IC (*see* figure above).

Hex

Lookup the '`3`' and toggle it in (7 bits, `b1` in the chart is bit 0). What is the *low nibble* digit showing? Do the same for the other values in the 0 to 9 range. What is the explanation for this?

Dec

Toggle in an uppercase '`A`'. What is its value in decimal? And for the lowercase '`a`'? What is the difference between the two decimal values? Notice it matches the annotation below the 'difference'-bit. Is this the same with '`W`' and '`w`'? Aha, this is how the *Shift*-key works!

© 2026, 8BitFlux.com

The layout of the keyboard you are using today still has traces of the original ASCII chart from 1972. Like *shifting* a '\' results in a '|', or the comma (,) being one bit away from the '<', to name a few.

The modern equivalent of ASCII is **Unicode**, especially the *UTF-8* standard is widely used. It is up to 4 bytes long and can represent all kinds of symbols: Chinese characters, Egyptian hieroglyphs and of course the *emoticons* we all like to use.

A smiley encoded in UTF-8 uses 4 bytes: F0 9F 98 9D.

You discovered that one BCD-digit was being 'contained' in the ASCII standard years ago. In 1992, ASCII itself got contained in UTF-8 to be backwards compatible.

Hex

A beautiful piece of **ASCII art** is shown below. How many *unique* characters are used in the first two lines? Use the Workbench and the ASCII chart to find out the hexadecimal representation of these characters encoded in ASCII. What are their *nibble* pairs?

```
                    .-.  .-')                .-')  _
                     \   ( OO )              (  OO) )
  .------.             ;------.\    ,-.-')  /      '._
 /   .-.   \    .-')   | .-.   |    |   |OO)|'--...__)
|    \_.' /   _(   OO) | '-' /_)   |   |   \'--.   .--'
 /   .-.  '.(,------.| .-.  `.    |   |(_/    |   |
```

Bits $b_7 b_6 b_5$ →	000	001	010	011	100	101	110	111
b_4 b_3 b_2 b_1 ↓ 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

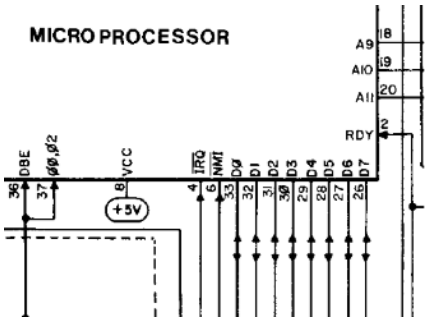
Early ASCII chart from 1972, is a BCD digit contained in it somewhere?

© 2026, 8BitFlux.com

6: Communicating Data

Remember *Morse code* from lesson 4? In this example the 'data' is transmitted in **serial** form. This means one 'bit' (a *short* or *long* beep) is sent (and received) after another, a sequential flow of signals.

Now imagine three sailors standing on deck of the same boat shouting all one letter in 'SOS' and doing this all at the same time. Pretty strange and inconvenient, but it is a way of **parallel** transmission of the same message.



Parallel communication used internally: the data bus (D0-D7) of the Apple-1.

Take a look at the Workbench and its *Schematics*. The toggle switches are the input to the microprocessor. Is this a *serial* or *parallel* way of data communication?

Data is communicated *serially* when it has to travel longer distances (*see* figure below). This way less electrical wires are needed, meaning lower cost. There is also immunity to *crosstalk*, i.e. adjacent wires do not interfere with each other.

Look back at the previous lesson, ASCII has some **control codes** that can be used to 'handshake' a serial transmission between two parties.

Take *enquiry* (`ENQ`), *acknowledge* (`ACK`) and *negative acknowledge* (`NAK`) for example. The handshake could be: a sender asking `hello are you there?` (`ENQ`) and a receiver answering with `yeah sure go ahead` (`ACK`) or `nah I am busy now` (`NAK`). After that, the actual data can be transferred.

The Pennywhistle modem: serial data transmission over telephone lines (Popular Electronics, March 1976).

Imagine being a *sender* and the message "`I'm OK`" has to reach the receiver, all encoded in ASCII. The handshake has been done, the last thing you heard from the other party was `6` (an ASCII control code in decimal). Is the receiver ready to receive your message?

Dec

Hex

The message is wrapped by a *start of text* (`STX`) and an *end of text* (`ETX`), followed by an *end of transmission* (`EOT`) control code. What full string of bytes has to be sent now, in hexadecimal nibbles?

© 2026, 8BitFlux.com

Things could go wrong during communication. A bit being a `1` suddenly becoming a `0` during transmission can have serious consequences. The use of **parity bits** is a basic mechanism to detect these kind of errors, i.e. the message received differs from the message that was sent.

ASCII used 7 bits originally, meaning one bit was 'free' in the byte. This bit (the *most significant* one) was used often as parity bit.

To do so, both of the parties (sender an receiver) have to agree on the variant to use: **even** or **odd parity**.

Calculating the parity bit comes down to first counting the total *amount of binary 1's* in the bit-string that is sent. The parity bit ensures that the total count, including the parity bit itself (being another `0` or `1`) is either *even* or *odd*. Examples are shown in the table.

		Count	Parity bit	
ASCII	**7 bits**	**of 1's**	**even**	**odd**
NUL	0000000	0	0	1
8	0111000	3	1	0
B	1000010	2	0	1
~	1111110	6	0	1
DEL	1111111	7	1	0

Calculation of the even or odd parity bit for a 7-bit ASCII character.

Dec Toggle the *least significant bit* to `0` and leave it there. Now toggle the other bits randomly and see the decimal numbers being displayed. What do you observe? Do the same with the LSb-bit toggled to `1`.

Hex In the previous assignment you wrote down the ASCII nibbles for the message "`I'm OK`". Rewrite them with an *even* parity bit added.

Hex In one occasion "`I'k O{`" was received. The same parity checking was used and no error was detected. How could this happen?

Oct Look back at lesson 2 with the assembly source listing of the Apollo computer (AGC). The column just right of the actual instruction (*dashed line*) contains only 0's and 1's. This was in fact a *parity bit* used by the AGC to check the integrity of the instruction read from memory. Did they use *even* or *odd* parity?

7: Into Negativity

It is freezing outside, brrr… Some applications need to work with negative numbers. Called negative **integers** without the fractional part. In mathematics negativity is just adding a minus sign. So how does this work in computing?

© 2026, 8BitFlux.com

Again we need some kind of *encoding* (*see* lesson 4). There are multiple methods to encode these so-called **signed** integers. The opposite of those are **unsigned** integers, the 'positive' numbering we did until now.

The most natural way is to just have one bit take care of negativity, without changing anything else. For a byte the *most significant bit* becomes a **sign-bit**. It is just a flag indicating the number (the *magnitude*) having a minus sign in front of it yes (1) or no (0). This is called the **sign-magnitude** representation.

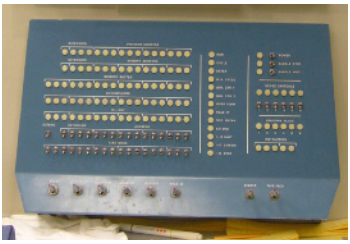
Note that the Workbench can not interpret *signed* integers, values are always shown as *unsigned* by the digits.

Dec

Bit `7` is the sign bit when using sign-magnitude. So we can't use that bit to store a number. What is the maximum *positive* decimal number that can be stored this way? And the minimum *negative*? What happens to the zero when the sign-bit is turned on?

GENERAL FORM	EXAMPLES
1 to 5 decimal digits. A preceding + or − sign is optional. The magnitude of the constant must be less than 32768.	3 +1 −28987

The IBM 704 mainframe computer used sign-magnitude, out of the Fortran Programmer's Reference Manual from 1956.



PDP-1 front panel, it used ones' complement. Photo (cropped) by Takuya Oikawa CC BY-SA 2.0.

Besides the weirdness of a *minus zero*, the method of sign-magnitude has another drawback. A more complex electrical circuit (within the processor) is needed in order to do addition *and* subtraction.

To solve this, **complements** can be used. Imagine the odometer from lesson 0 running backwards: `0001`, `0000`, `9999`, `9998` and so on. This is how to count into the negative using complements, the number 9998 corresponds to -2, the 8 here is called the *ten's complement* of 2. Now calculate 5 + 9998 = 10003. Ignore the 1 and the 3 is the answer to 5 - 2. This is how **subtraction by addition** works.

Slide the *Flip the bits*-switch to *Ones' Complement*. In lesson 1 you counted from 0 to 9 in BCD. Do the same now and observe the decimal number counting down, like an odometer moving backwards. And can you create a *minus zero*? What range can the numbers be?

Dec

It is easier for a computer (and its circuit) when *complements* are being used, as it can just stick to performing addition only.

© 2026, 8BitFlux.com

See the table: -2 + 2 has to be 0, but results in `-0` for the **ones' complement** (going up two rows). The minus zero is being inconvenient here.

To fix this, a 1 can be added to the flipped bits. This creates the **two's complement** and the -0 disappears. The minimum negative number 'grows' to `-128`, instead of being `-127` for the other signed representations.

In modern computing of today, the two's complement method is used to represent signed integers.

Decimal	Unsigned	Sign-magnitude	Ones'	Two's
1	0001	**0**001	**0**001	**0**001
0	0000	**0**000	**0**000	**0**000
-0	-	**1**000	**1**111	-
-1	-	**1**001	**1**110	**1**111
-2	-	**1**010	**1**101	**1**110

Different 4-bit representations of an integer. The sign-bit is in bold.

Dec

Only use the *low nibble* and set the *Flip the bits*-switch to *Normal*. **Press and hold** the mode button until the digits turn off. *See* the table above, which bit number is the sign-bit? Toggle in the maximum *positive* bit-value that can be kept by the 4-bit wide two's complement integer? What is this in decimal? Hold the mode button again to see if you are correct.

Dec

Do not change the toggle switches and turn the digits off again. Move the *Flip the bits*-switch to *Ones' Complement* and turn the LEDs of the *high nibble* off. Observe only the sign-bit is 'on'. Look at the table, what *negative* decimal number would be represented by `1000` for two's complement? And how will the Workbench show it, thinking it is an unsigned integer? Hold the button again and find out.

Play the Game

Press the **Game**-button to start a new game for the *active* Mode, being: *Decimal*, *Hexadecimal* or *Octal*. After '`Go`' the green LED stops blinking and a random value (in one of the three number systems) is shown. Guess the value by toggling the bits. The quicker you are, the more points you earn. Exit the game by pressing the Mode-button again.

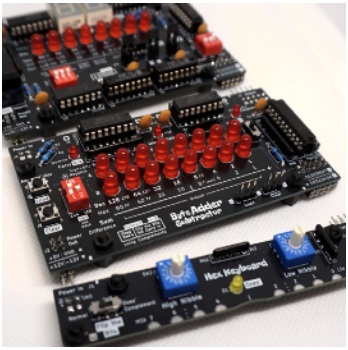
A high score is kept for each number system and is shown after '`Hi`' when a game starts. The scoring is *always* in decimal. Improve your skill at the game by doing the lessons, and you'll figure out when the *final boss* is to be expected. **Press and hold** the Game-button for 5 seconds to reset *all* high scores.

© 2026, 8BitFlux.com

Modular 8-bit Workbench™

Want to learn more (by doing) about computing and electronics? Check out the *non*-ED **8-bit Workbench™** modular platform. Here, the *output* (the binary LEDs and digit display) is separated from the *input* (**Binary** or **Hex Keyboard**) and you can choose boards to mix and match in-between functionality:

- Take a **Super Breadboard 2** and wire your own *short-circuit protected* logic gates (transforming an input) or connect with your own (vintage) computer. Hook up with other electronic platforms that involve some programming, like an *Arduino*, *micro:bit* or *ESP32*.
- Experiment with the famous 555-timer IC using the **Terrific Timer**, let it be your adjustable clock or create a fast binary counter. Learn about *latching*, *astable* and *monostable* concepts.
- Dive into arithmetic and the workings inside of a processor with the **Byte Adder Subtractor**, an 8-bit full adder with *carries* and a built-in *accumulator*.
- More ideas? Let us know!



Vist the website 8BitFlux.com for more info.

Acknowledgements

Many thanks goes out to *Joe Ochwat* (writer at COMPUTE! Publications, Inc.) and *Bart van den Akker* (computer historian at HomeComputerMuseum) for their feedback.

The in-game text messages are in the Siekoo-Alphabet by *Alexander Fakoo*, for info: fakoo.de/en/siekoo. This is a 100% original human-generated manual and may not be used to train AI.

DISCLAIMER

The use of our products and schematics is at your own risk. We expect our audience and buyers to have experience with electronics and electricity. The product images shown on the website are for illustration purposes only and may not be an exact representation of the actual product, which is subject to enhancement. Colors (of parts) can differ because of changes in supply.

In no event shall 8BitFlux.com be liable in warranty, contract, negligence, strict liability, tort or otherwise, for any damages, whether incidental or consequential, which are alleged to be caused by one or more of our products, beyond the cost to the purchaser of the subject product or products in question.

© 2026, 8BitFlux.com