Beginner's Programming Tutorial in QBasic

This document is meant to get you started into programming, and assumes you have some experience with computers and with Windows 95 (or 98, etc.).

Since this tutorial is written for people who don't like to read a lot of text, it includes a number of examples. Therefore, you can do a lot of *work* in not much *time*.

The more important chapters have a star ().

http://www.geocities.com/progsharehouse/qbtutor
Mirror: http://development.freeservers.com/qbtutor

Table of Contents

Part I: Q-Basics

Part II: Intermediate topics

Part III: Advanced topics

Before you start

Before you can create a program in QBasic, you need the ***QBasic interpreter***. It is available from your Windows 95 (or 98) CD, or you can download it below.

To access QBasic from the **Windows 95** CD:

1. Insert the CD into your CD-ROM drive.
2. Click **"browse this CD"** (if the menu screen doesn't come up, then browse the CD from **My Computer**.
3. Go to the **\OTHER\OLDMSDOS** directory.
4. Open a program called QBASIC.EXE (this is version 1.1 of the QBasic interpreter).

To access QBasic from the **Windows 98** CD:

1. Insert the CD into your CD-ROM drive.
2. Click **"browse this CD"** (if the menu screen doesn't come up, then browse the CD from **My Computer**.
3. Go to the **\TOOLS\OLDMSDOS** directory.
4. Open a program called QBASIC.EXE (this is version 1.1 of the QBasic interpreter).

Download it here (**right-click** and press **"Save As")**:

QBASIC.ZIP (323 KB) - QBasic 1.1 interpreter and sample programs

UNZIP32.EXE (90 KB) - Extracts the ZIP file

To unzip the QBASIC.ZIP file with UNZIP32.EXE:

a. Go to the **Start Menu**

b. Click **Run...**

c. Type the following (this loads MS-DOS):

```
command  <Enter>
```

d. Enter the following in DOS (assuming you saved QBASIC.ZIP to **C:\QBASIC**):

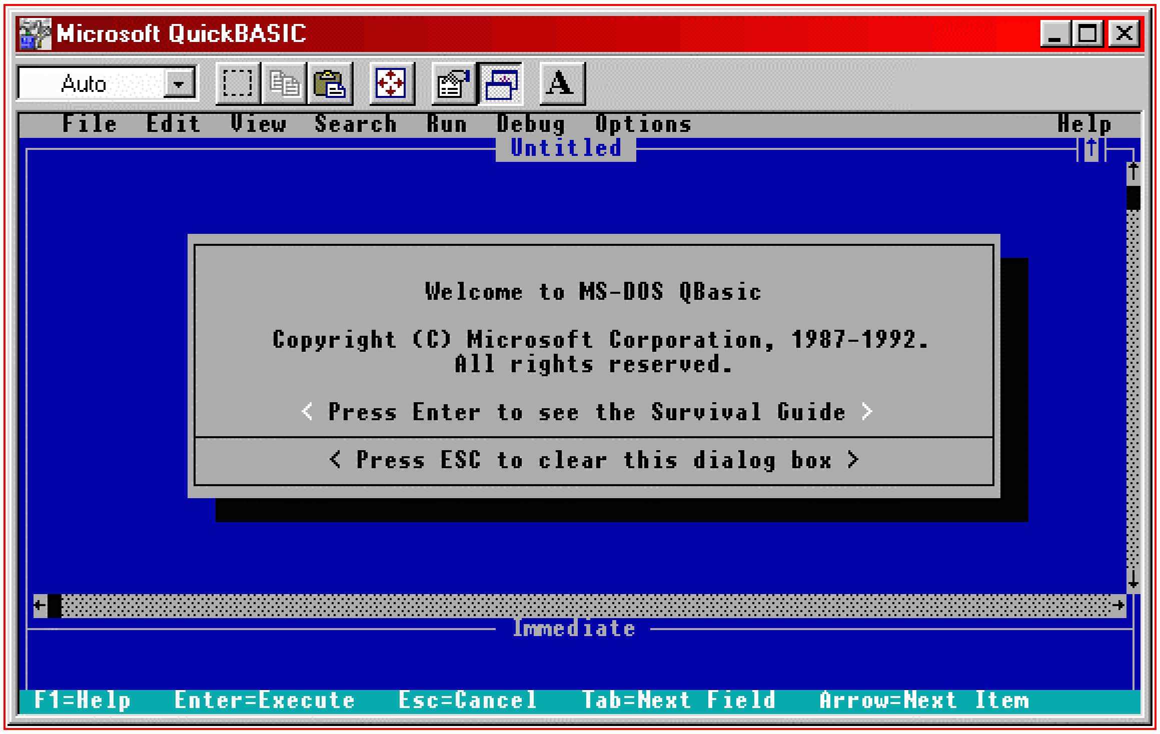
```
cd c:\qbasic
unzip32 -n qbasic.zip
```

Your first program

After launching the QBasic interpreter (see *before you start*), you might see a window requesting a list of "parameters." If this window comes up, press the `Enter` key to continue.

You should now see the QBasic interpreter, which has a blue background and displays a dialog box at the center. (If the interpreter fills the entire screen, then you may want to press `"Alt + Enter,"` to make it smaller.)

Press the `Esc` key to hide the dialog box.



QBasic interpreter - main screen

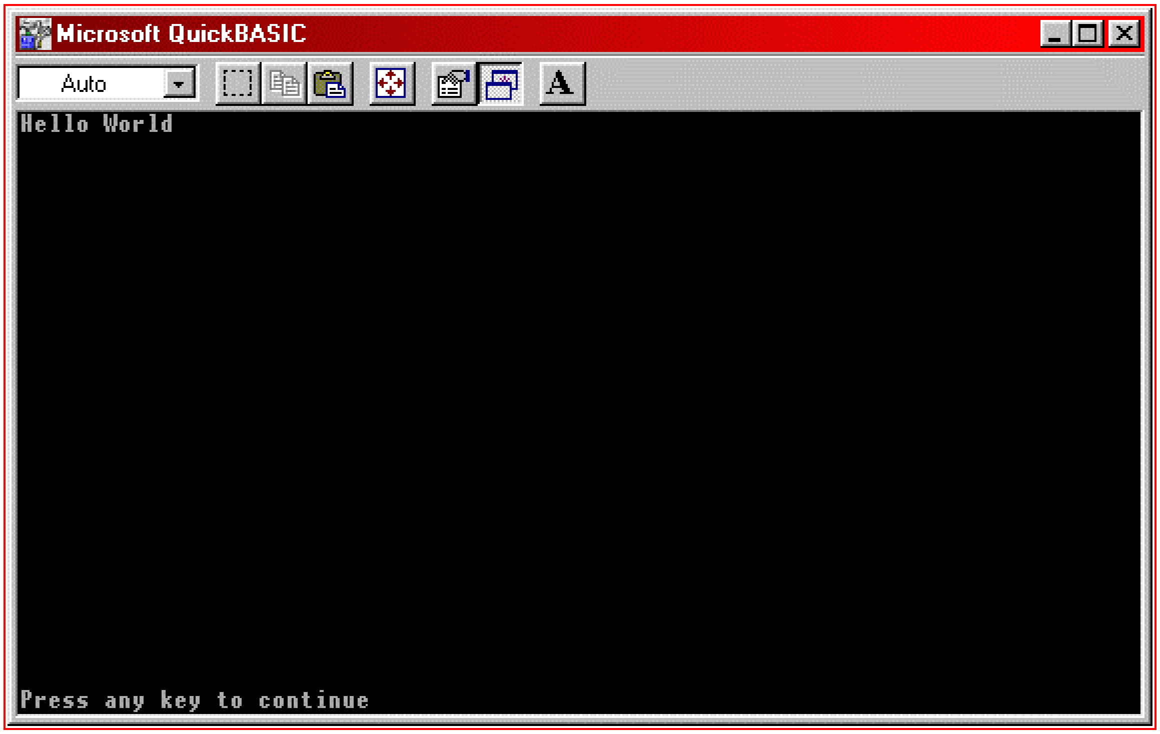
Type the following (including the quotation marks) in the QBasic interpreter:

```
PRINT "Hello World!"  <press Enter>
```

Now press `F5` to run the program. You should now see a black screen, with `Hello World` at the top, and `Press any key to continue` at the bottom.

Press a key on the keyboard to return to the main screen.

(The figure below displays the **"output screen."**)



QBasic interpreter - output screen

If you run the program again, the interpreter adds another `Hello World`. QBasic adds `Hello World` each time the program is run.

Deleting the program

To erase the current program:

1. Go to the "File" menu.
2. Click "New."
3. The interpreter asks if you want to save the program.
4. Select "No" (or if you'd rather keep the program, select "Yes").

Strings

There are certain types of data (or information) called "strings." Strings contain a sequence of characters (letters, numbers, and symbols) enclosed in quotation marks. For example, `"Hello World!"` is a string.

The following is also strings:

```
"0123456789"

"This is a string"

"abc123"

"1 + 1 = 2"

"!@#$%^&*()"
```

Commands

There are also special functions called "commands" (also called "instructions"). A "command" tells the QBasic interpreter to do something.

The **`PRINT`** command tells the QBasic interpreter to print something to the screen. In this case, the interpreter printed `"Hello World!"`.

> **TIP:** Instead of typing `PRINT`, you can enter a question mark. For example:
>
> **?"Hello World!"**

With the `PRINT` command, you can also print **numbers** to the screen. Delete the current program (unless you already have) and write the following:

```
PRINT 512 (or ?512)

<press Enter>
```

Press `F5` to run the program. The program outputs:

```
512
```

Expressions

An expression is something the interpreter calculates (or evaluates). Such as:

```
1 + 1              (returns 2)

100 - 47           (returns 53)

3 * 34             (returns 102)

80 / 4             (returns 20)

(100 * 3) + 56     (returns 356)
```

NOTE: The asterisk `(*)` means to multiply two numbers; the slash `(/)` means to divide

If you pass an expression to the `PRINT` command, the value returned (a number) is printed.

Clear the current program, and then run the following:

```
PRINT 512 + 478
```

Program output:

```
990
```

If you enclose the expression with quotation marks, the expression becomes a string and isn't evaluated. For example:

```
PRINT "512 + 478"
```

Output:

```
512 + 478
```

TIP: To clear the output screen, use the CLS command.

CLS

More about the PRINT command

You can use **multiple** print statements in your program.

```
PRINT "Hello"
PRINT "World"
```

Output:

```
Hello
World
```

To place `World` onto the previous line, place a semi-colon after `PRINT "Hello"`.

```
PRINT "Hello";
PRINT "World"
```

Output:

```
HelloWorld
```

Also, if you put a comma instead of a semi-colon on the first line, the program will insert spaces between the two words.

```
PRINT "Hello",
PRINT "World"
```

Output:

```
Hello World
```

Variables

This chapter discusses an important topic in programming, ***"variables."*** Please read this section thoroughly.

A variable is a piece of data kept in the computer's memory (RAM). The *location* of a variable in RAM is called the *"address."*

	Memory address	Value
	999997	0
	999998	0
	999999	0
Variable →	1000000	0
	1000001	0
	1000002	0
	1000003	0
	.	.
	.	.
	.	.

How a variable is stored in RAM

The following program prints the variable X to the screen:

```
print X
```

Since the variable hasn't been assigned a number, the value of the variable is 0. So, the output of the program is:

```
0
```

This next program sets X to 15, and then prints the variable:

```
X = 15

print X
```

This time, the output is:

```
15
```

In the above example, the number 15 was stored in the computer's RAM at a certain memory address. Then the **PRINT** command accessed (or looked at) that address when it printed "15" to the screen.

	Memory address	Value
	999997	0
	999998	0
	999999	0
Variable X ──►	1000000	15
	1000001	0
	1000002	0
	1000003	0
	.	.
	.	.
	.	.

(**NOTE:** The memory address of `X` is not necessarily `1000000`)

> **ADVANCED TIP:** Although you don't normally need to, you can find the actual memory address of a variable (`X`, for example) by using the `VARSEG` and `VARPTR` commands.
>
> **PRINT (VARSEG(X) * 65536) + VARPTR(X)**
>
> (For more information, see *Memory*.)

As in the programs above, a variable is accessed by calling its name. Variable names can have a combination of letters and numbers. The following are valid variables:

```
Y

num

VALUE

xYz

abc123
```

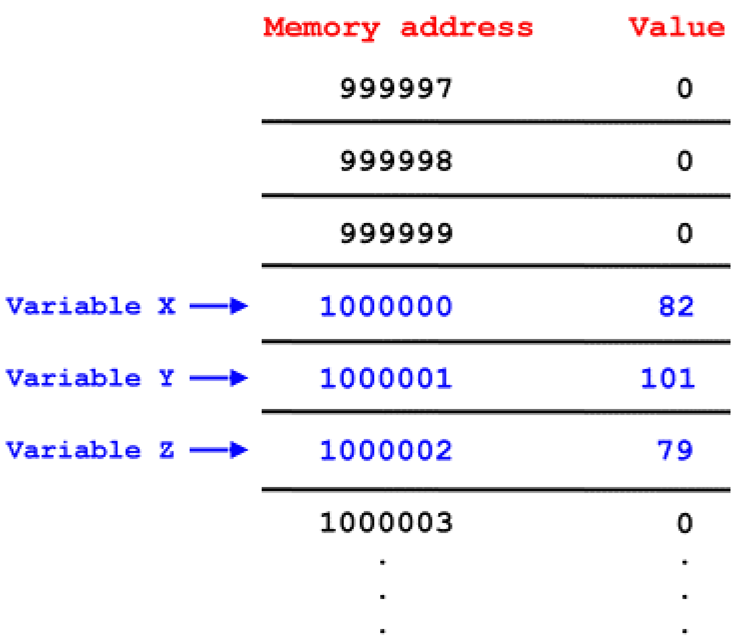
Also, you can use *multiple* variables in your program.

```
X = 82
Y = 101
Z = 79

PRINT X
PRINT Y
PRINT Z
```

Output:

```
82
101
79
```



(**NOTE:** The memory addresses of these variables are not necessarily as specified)

Expressions

If you pass an expression to a variable, the expression is evaluated and the variable is set to that value.

```
x = 500 + (10 * 7)

PRINT x
```

Output:

```
570
```

You can also use *variables* as expressions.

```
rate = 50
time = 2

distance = rate * time

PRINT distance
```

Output:

```
100
```

Plus, you can have both variables and numbers in an expression.

```
X = 100
Y = X * 7

PRINT Y
```

Output:

```
700
```

TIP: The following increases `X` by 1:

X = X + 1

Strings

If you add a dollar sign `($)` to the end of a variable, the variable is a string.

```
X$ = "Hello World!"

PRINT X$
```

Output:

```
Hello World!
```

If you try to set a string to a non-string variable, an error occurs.

```
X = "Hello World!"
```

The QBasic interpreter says `"Type mismatch"` when you try to run the above program.

A string can be added to the end of an existing variable string.

```
X$ = "Hello"
X$ = X$ + "World"

PRINT X$
```

Output:

```
HelloWorld
```

You can also add variable strings together.

```
a$ = "String1"
b$ = "String2"
c$ = "String3"

d$ = a$ + b$ + c$

PRINT d$
```

Output:

```
String1String2String3
```

Retrieving keyboard input from the user

One way to receive input from the keyboard is with the `INPUT` command. The `INPUT` command allows the user to enter either a string or a number, which is then stored in a variable.

```
INPUT data$

PRINT data$
```

When this program is executed, the `INPUT` command displays a question mark, followed by a blinking cursor. And when you enter text, the program stores that text into the variable `data$`, which is printed to the screen.

> **TIP:** If you place a string and a semi-colon between `INPUT` and the variable, the program will print the string.
>
> **INPUT "Enter some text:"; data$**

To receive a number, use a non-string variable.

```
INPUT number

PRINT number
```

If you enter text instead of a number, the QBasic interpreter displays an error message `("Redo from start")`.

Below is another example of the `INPUT` command:

```
PRINT "Enter some text:"
INPUT text$

PRINT "Now enter a number:"
INPUT num

PRINT text$
PRINT num
```

> **TIP:** You can have the question mark displayed on the previous line by using a semi-colon.
>
> **PRINT "Enter some text:";**
> **INPUT text$**

The IF and THEN commands

The `IF` and `THEN` commands are used to compare an expression and then perform some task based on that expression.

```
x = 5

IF x = 5 THEN PRINT "x equals 5"
```

Since `X` does equal 5 in this case, the program outputs:

```
x equals 5
```

Expression signs

You can also enter the following statements, instead of the equals sign:

```
x < 5 (x is less than 5)

x > 5 (x is greater than 5)
```

Run the following:

```
x = 16

IF (x > 5) THEN PRINT "x is greater than 5"
```

Output:

```
x is greater than 5
```

You can also combine the signs like this:

```
x <= 5 (x is less than or equal to 5)

x >= 5 (x is greater than or equal to 5)

x <> 5 (x does not equal 5)
```

Run the following example:

```
CLS

x = 5

IF (x >= 5) THEN PRINT "x is greater than or equal to 5"
IF (x <= 5) THEN PRINT "x is less than or equal to 5"
IF (x <> 5) THEN PRINT "x does not equal 5"
```

Output:

```
x is greater than or equal to 5
x is less than or equal to 5
```

ELSE

Using the `ELSE` command, you can have the program perform a *different* action if the statement is false.

```
x = 3

IF x = 5 THEN PRINT "Yes" ELSE PRINT "No"
```

Since `X` doesn't equal 5, the output is:

```
No
```

END IF

`END IF` allows you to have multiple commands after the `IF...THEN` statement, but they must start on the line *after* the `IF` statement. `END IF` should appear right after the list of commands.

```
x = 5

IF (x = 5) THEN

   INPUT a$
   PRINT a$

END IF
```

The following program uses `ELSE` with the `END IF` command:

```
x = 16

IF (x = 5) THEN

   INPUT a$
   PRINT a$

ELSE

   PRINT x * 2

END IF
```

Output:

```
32
```

> **TIP:** There is a way to have **multiple** commands after `IF...THEN` without using `END IF`. To do so, place a colon between each command.
>
> **IF (x = 5) THEN INPUT a$: PRINT a$**

ELSEIF

The `ELSEIF` command allows you to perform a secondary action if the first expression was false. Unlike `ELSE`, this task is only performed if a specified statement is true.

```
x = 6

IF (x = 5) THEN
   PRINT "Statement 1 is true"

ELSEIF (x = 6) THEN
   PRINT "Statement 2 is true"

END IF
```

Output:

```
Statement 2 is true
```

You can have multiple `ELSEIF` commands, along with `ELSE`.

```
x = 8

IF (x = 5) THEN
   PRINT "Statement 1 is true"

ELSEIF (x = 6) THEN
   PRINT "Statement 2 is true"

ELSEIF (x = 7) THEN
   PRINT "Statement 3 is true"

ELSE
   PRINT "No above statements are true"

END IF
```

Output:

```
No above statements are true
```

Multiple expressions

You can have more than one expression in `IF...THEN` by using either the `OR` operator or the `AND` operator.

The `OR` operator only requires one expression to be true in order to print "Yes" in the following program:

```
x = 20

IF (x = 5 OR x = 20) THEN PRINT "Yes"
```

Output:

```
Yes
```

The `AND` operator requires both expressions to be true.

```
x = 7

IF (x > 5 AND x < 10) THEN PRINT "True"
```

Output:

```
True
```

This is a slightly more complex example:

```
x = 16
y = 3

IF ((x > 5 AND x < 10) OR y = 3) THEN PRINT "Correct"
```

Output (since `Y` is 3):

```
Correct
```

Strings in IF...THEN

So far in this chapter, we've only been dealing with numbers, but you can also use strings with the `IF...THEN` command.

```
x$ = "Hello"

IF (x$ = "Hello" OR x$ = "World") THEN PRINT x$
```

Output:

```
Hello
```

You can also compare two variable strings:

```
x$ = "Hello"
y$ = "World"

IF (x$ <> y$) THEN PRINT x$; " "; y$
```

Output:

```
Hello World
```

Labels and the GOTO and GOSUB commands

The GOTO and GOSUB commands enables you to jump to certain positions in your program. Labels are used to specify what point in the program to continue execution.

GOTO

To use GOTO, place a label somewhere in your program, and then enter.

```
GOTO <label>
```

Run the following example program:

```
PRINT "1"

GOTO TheLabel

PRINT "2"


TheLabel:

PRINT "3"
```

Output (notice how PRINT "2" is skipped):

```
1
3
```

> **TIP:** TheLabel can be placed on the same line as PRINT "3"
>
> **TheLabel: PRINT "3"**

GOSUB

The GOSUB command is the same as GOTO, except when it encounters a RETURN statement, the program "returns" back to the GOSUB command. In other words, RETURN continues program execution immediately after the previous GOSUB statement.

```
PRINT "1"

GOSUB TheLabel

PRINT "2"

END
```

```
TheLabel:

PRINT "3"

RETURN
```

(Note: The `END` command exits the program.)

Since the program returns to the `GOSUB` command, the number 2 is printed this time.

```
1
3
2
```

Line numbers

"Line numbers" can be used as labels.

```
PRINT "1"

GOTO 10

PRINT "2"

10 PRINT "3"   (Notice the line number)
```

You can also write the program like this:

```
10 PRINT "1"

20 GOTO 40

30 PRINT "2"

40 PRINT "3"
```

The line numbers don't even have to be in sequence.

```
17 PRINT "1"

2 GOTO 160

701 PRINT "2"

160 PRINT "3"
```

Each of these programs output:

```
1
3
```

Guessing game

The following is a simple guessing game:

```
CLS

start:
PRINT "Guess a number between 1 and 10: ";
INPUT num

IF (num < 1 OR num > 10) THEN
   PRINT "That is not between 1 and 10"
   GOTO start
END IF

IF (num = 6) THEN
   PRINT "Correct!!!"
ELSE
   PRINT "Try again"
   PRINT
   GOTO start
END IF
```

Output (may be slightly different):

```
Guess a number between 1 and 10: ? 2
Try again

Guess a number between 1 and 10: ? 7
Try again

Guess a number between 1 and 10: ? 6
Correct!!!
```

TIP: Notice the second `PRINT` statement under `PRINT "Try again"`. It adds a blank line under `Try again` when the program is executed.

Loops

"Loops" make it easier to do an action multiple times. There are at least four types of loops: `IF...GOTO`, `WHILE...WEND`, `DO...LOOP`, and `FOR...NEXT`.

IF...GOTO

This program uses `IF...GOTO` to create a loop:

```
x = 10

start:
PRINT x

x = x + 1  (This adds 1 to x)

IF x < 15 THEN GOTO start
```

Output:

```
10
11
12
13
14
```

WHILE...WEND

The `WHILE...WEND` commands continue a loop until a specified expression is false.

To use `WHILE...WEND`:

1. Place an expression after WHILE

2. Enter a list of commands

3. Place WEND at the end

Run the following:

```
x = 10

WHILE x < 15

   PRINT x

   x = x + 1

WEND
```

Output (same as in previous example):

```
10
11
12
13
14
```

DO...LOOP

`DO...LOOP` is exactly the same as `WHILE...WEND`, except it has at least two slight advantages. With `DO...LOOP` you can:

1. Loop until an expression is true

2. Loop at least one time regardless of whether the expression is true or not.

To use `DO...LOOP`:

1. Specify whether the loop continues "while" the expression is true or "until" the expression is true, using the WHILE and UNTIL statements, respectively.

2. Place an expression after WHILE/UNTIL

3. Enter a list of commands

4. Place LOOP at the end

The following uses the `WHILE` statement:

```
x = 10

DO WHILE x < 15

   PRINT x

   x = x + 1

LOOP
```

This program uses the `UNTIL` statement:

```
x = 10

DO UNTIL x = 15

   PRINT x

   x = x + 1
```

```
LOOP
```

They both output:

```
10
11
12
13
14
```

If you place the expression at the end of the loop instead, the program goes through the loop at least once.

```
x = 32

DO

   PRINT x

   x = x + 1

LOOP WHILE x < 5
```

This is the output because the loop was only gone through one time:

```
32
```

FOR...NEXT

`FOR...NEXT` provides an easier way to create a loop.

```
FOR x = 1 TO 5

   PRINT x

NEXT x
```

Output:

```
1
2
3
4
5
```

> **TIP:** The `X` after `NEXT` is optional (unless you have a loop within a loop).

Also, you can use the `STEP` attribute to specify how much `X` will be increased each time through the loop.

```
FOR x = 1 TO 5 STEP 2

   PRINT x

NEXT x
```

Output:

```
1
3
5
```

STOPPING LOOPS

To stop a loop prematurely, use the `EXIT` command, followed by either `FOR` or `DO`.

```
FOR x = 1 TO 5

   PRINT x

   IF (x = 3) THEN EXIT FOR

NEXT x
```

Output:

```
1
2
3
```

(**NOTE:** This command only works with the `DO...LOOP` and `FOR...NEXT` commands, not with `WHILE...WEND` or `IF...GOTO`.)

What next?

Congratulations! You've finished part 1 of this tutorial. The remaining chapters cover additional topics, and don't have to be read in sequence (one after another).

If you want, you can move on to a more advanced programming language. The rest of this chapter briefly explains how you can start using the most popular ones.

One reason you may want to move on, at least at some point, is because QBasic has minimal capabilities. One example of this is that you can't create executable programs (EXE files) in QBasic. (*QuickBasic 4.5* can create these files, but this product is no longer on the market.)

Other programming languages

Information about the below programming languages can be found at ProgrammingTutorials.com.

C and C++

You can learn C by going to www.cm.cf.ac.uk/Dave/C/CE.html, www.strath.ac.uk/CC/Courses/NewCcourse/ccourse.html, or www.programmingtutorials.com/tutorial.asp?id=C.

Before you can create an EXE file in C, you must have a compiler. I recommend downloading the DJGPP compiler (www.djgpp.com). This program is free, however, the author does accept donations.

See below for information on getting DJGPP.

You may also want to get the Allegro programming library. This library is useful for creating games in C.

(**NOTE:** C++ is a more powerful version of C. It introduces "object oriented" programming. I recommend learning C++ before moving on to Visual C++.)

Visual C++

With Visual C++, you can create Windows 95 programs, instead of DOS. It costs about $100 for the standard version.

For more information, click here.

To purchase Visual C++, click here.

Visual Basic

Visual Basic is similar to QBasic. So, if you are highly involved in QBasic, then you may want to switch directly to Visual Basic, instead of learning C/C++ or Visual C++.

With Visual Basic, like Visual C++, you can create Windows 95 programs. It costs about $100 for the "learning" edition.

For more information, click here.

To purchase Visual Basic, click here.

Getting DJGPP

Since the installation instructions for DJGPP are a little confusing, I've provided my own below.

For more information about DJGPP, visit the DJGPP website at www.delorie.com/djgpp.

Installing DJGPP

1. Create a new folder called **DJGPP** in drive **C**.

2. Download and save each of the following *to the **DJGPP** folder:*

   (If any of the links below are out of date, or if you want to download more DJGPP packages, you can find the most current files here.)

   bnu211b.zip (2.6 MB) - Basic assembler, linker

   djdev203.zip (1.5 MB) - DJGPP Basic Development Kit

   faq230b.zip (0.6 MB) - Frequently Asked Questions

   gcc2953b.zip (1.9 MB) - GCC compiler

   gpp2953b.zip (1.7 MB) - C++ compiler

   mak3791b.zip (0.3 MB) - Make (processes makefiles)

   rh1478b.zip (2.0 MB) - RHIDE, an Integrated Development Environment, has a built-in editor and debugger

   txi40b.zip (0.6 MB) - Info file viewer

   unzip32.exe (0.1 MB) - Extracts the ZIP files

3. Unzip each of the files to the **DJGPP** folder.

   If you are using UNZIP32.EXE:

   a. Go to the **Start Menu**

b. Click **Run...**

c. Type the following (this loads MS-DOS):

```
command  <Enter>
```

d. Enter the following in DOS:

```
cd c:\djgpp

unzip32 -n *.zip
```

4. Edit your **AUTOEXEC.BAT** file.

    a. Go to the **Start Menu**

    b. Click **Run...**

    c. Type the following (you can **"copy and paste"** it):

    ```
    notepad c:\autoexec.bat
    ```

    d. Add the following lines to the **end** of the file (you can **"copy and paste"** this also):

    ```
    set DJGPP=C:\DJGPP\DJGPP.ENV
    set PATH=C:\DJGPP\BIN;%PATH%
    ```

    e. Go to the **File** menu.

    f. Click **Save**.

    g. Close the program.

5. Restart your computer.

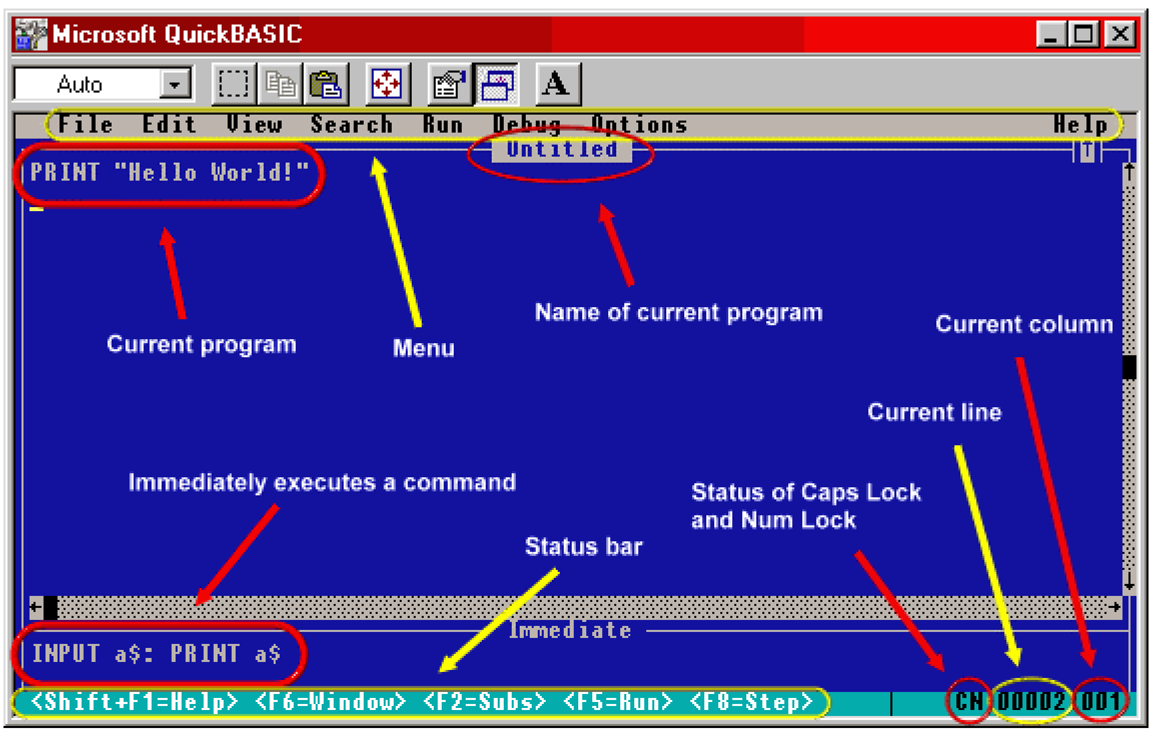
6. Run a program called **RHIDE.EXE** in the **c:\djgpp\bin** folder.

QBasic interface

This chapter gives a brief overview of the QBasic interface.

The interface has the following features:

1. Window displaying the current program
2. Menu
3. Name of current program
4. Window to immediately execute a command
5. Status bar
6. Status of Caps Lock and Num Lock
7. Current line of cursor
8. Current column of cursor



QBasic interface

Current program

The current program is displayed in the middle of the screen, and covers most of the QBasic interface.

Menu

The menu provides most of the operations for the QBasic editor. Such as opening a file, pasting text, and searching for a string.

File

New - Clears the current program

Open - Loads a program from disk

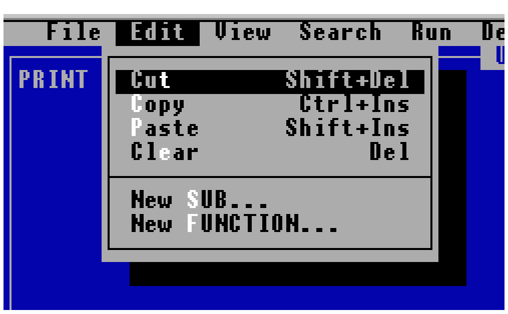
Save - Saves the current program to disk

Save As - Saves the program, but under a different name

Print - Prints the selected text, current window, or entire program

Exit - Closes the QBasic interpreter

Edit



Cut - Removes the selected text and stores it in the clipboard

Copy - Copies the text instead of removing it

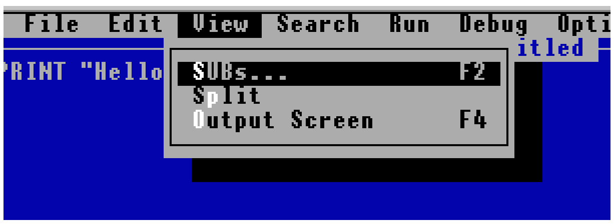
Paste - Adds the text in the clipboard to the current position of the cursor

Clear - Removes the text **without** storing it to the clipboard

New Sub - Enables you to create a new subroutine (see *Subroutines and Functions*)

New Function - Enables you to create a new function (see *Subroutines and Functions*)

View



SUBs - Shows the list of current subroutines and functions (see *Subroutines and Functions*)

Split - Displays the contents of the current program in two windows. If the window is already split, this hides the second window (NOTE: The text in each window is always the same, even if you alter the text in one window)

Output Screen - Shows the QBasic output screen.

Search

Find - Allows you to search for a string of text in the program

Repeat Last Find - Continues the previous search operation

Change - Replaces each instance of a string with another string

Run

Start - Executes the current program

Restart - Starts from the beginning

Continue - Continues execution at the current position

Debug

Step - Processes the next command

Procedure Step - Processes the next command, but does not show QBasic going inside a subroutine or function

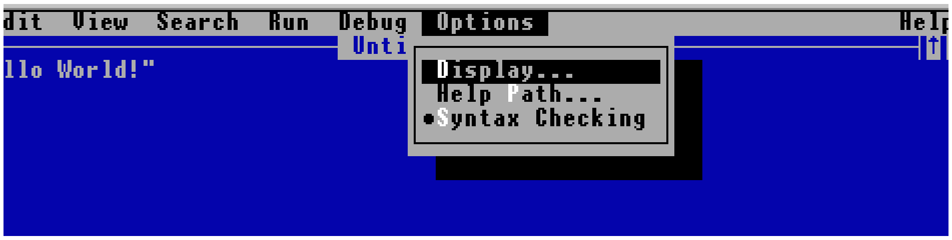
Trace On - Shows the command that is being executed while the program is running

Toggle Breakpoint - Sets or removes a breakpoint. Use this to have the QBasic interpreter stop when it reaches a specified line in the program

Clear All Breakpoints - Removes all breakpoints in the program

Set Next Statement - Allows you to continue execution at the specified line

Options

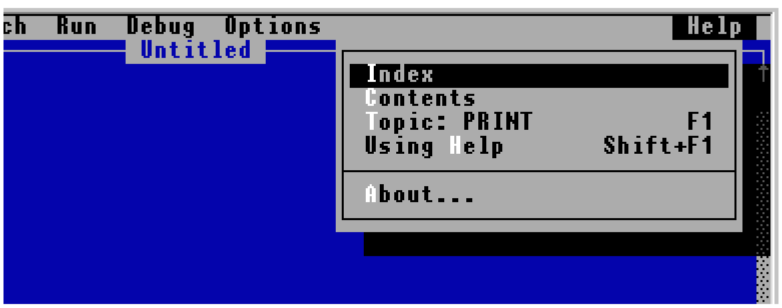


Display - Enables you to change display colors, the number of spaces to use for tabs, and whether or not scroll bars are visible

Help Path - The location of the QBASIC.HLP file

Syntax Checking - Allows you to have the QBasic editor check the syntax of your program as you type

Help



Index - List of all QBasic commands, keywords, operators, etc.

Contents - The table of contents for QBasic help

Topic - Show help for a specific keyword

Using Help - Displays information on using QBasic help

About - Shows information about the QBasic interpreter

Name of current program

The file name of the current program is displayed near the top of the screen in the center. You can change the name by selecting **"Save As"** on the **"File"** menu.

Immediately execute a command

QBasic provides a way to execute a command without running the current program. To do so, select the bottom window (under **"immediate"**) and enter a command, then press `Enter`.

Status bar

The status bar is at the bottom of the screen. It displays a short list commands
`(<Shift+F1=Help> <F6=Window> <F2=Subs> <F5=Run> <F8=Step>)`.

When you highlight an item on the menu, the status bar displays a short description of what the item does.

Status of Caps Lock and Num Lock

If **Caps Lock** is set, a **"C"** is displayed on the right side of the status bar.

If **Num Lock** is set, an **"N"** is displayed on the right side of the status bar.

Current line

On the right side of the status bar, the current **line** of the cursor is displayed.

Current column

On the right side of the status bar, the current **column** of the cursor is displayed (immediately after the current line).

Adding documentation to your programs

Documenting your program (also called "commenting") allows you to remind yourself about something in your program. Plus, if your program is seen by other people, documenting can help them understand your code.

The `REM` (remark) command enables you to add comments to your program without the text being treated like an instruction.

```
CLS
PRINT "Some text"

REM This text is ignored.

REM This program clears the output screen,
REM and then prints "Some text."
```

> **TIP:** You can use an apostrophe instead of the `REM` command.
>
> **' Comment goes here**

You can add `REM` to the same line as another command by placing a colon after the first instruction.

```
CLS:          REM This command clears the screen
PRINT "Text": REM This command prints "Text" to the screen
PRINT 534:    REM This prints the number 534 to the screen
```

NOTE: If you use an apostrophe instead of `REM` while doing this, you do not need to add a colon.

```
CLS           ' This command clears the screen
PRINT "Text"  ' This command prints "Text" to the screen
PRINT 534     ' This prints the number 534 to the screen
```

Reading and writing to files

To save data to a file:

1. Call the OPEN command, specifying the file name, file mode (OUTPUT), and file number.

2. Use PRINT, followed by the file number and the data you want to write.

3. Close the file using the CLOSE command.

The following opens a file, using mode `OUTPUT` and number 1, and then saves the text `Hello World!` to the file:

```
OPEN "testfile.dat" FOR OUTPUT AS #1
PRINT #1, "Hello World!"
CLOSE #1
```

To open a file for "reading," call `OPEN` and pass `INPUT` as the file mode. Then you can read the data by using the `INPUT` command.

```
OPEN "testfile.dat" FOR INPUT AS #1
INPUT #1, text$
CLOSE #1

PRINT text$
```

Output:

```
Hello World!
```

Displaying graphics

Before you can show graphics images on the screen, you must call the `SCREEN` command. `SCREEN` sets the graphics mode.

The following program uses graphics mode **13** (320x200) to display a line, then returns back to text mode:

```
SCREEN 13

' This starts at 10 pixels from the left, 10 from
' the top and goes to point (100, 100):

LINE (10, 10)-(100, 100)

WHILE INKEY$ = "": WEND  ' Waits until a key is pressed

SCREEN 0  ' Returns to text mode
```

You can also draw a colored line.

```
SCREEN 13

LINE (10, 10)-(100, 100), 192  ' Dark green

WHILE INKEY$ = "": WEND

SCREEN 0
```

To draw a single pixel, use `PSET`.

```
SCREEN 13

PSET (160, 100)

WHILE INKEY$ = "": WEND

SCREEN 0
```

The following displays a circle at **(160, 100)** with a radius of **50**:

```
SCREEN 13

CIRCLE (160, 100), 50

WHILE INKEY$ = "": WEND

SCREEN 0
```

Finally, to display a square, use `LINE`.

```
SCREEN 13
LINE (10, 10)-(100, 100), 192, B  ' Notice the B
WHILE INKEY$ = "": WEND
SCREEN 0
```

Mathematics functions

QBasic provides several functions to do mathematical calculations. A few of them are discussed here.

SQR

Use `SQR` to find the **"square root"** of a number.

```
PRINT SQR(1)
PRINT SQR(4)
PRINT SQR(9)
PRINT SQR(16)
PRINT SQR(25)
```

Output:

```
1
2
3
4
5
```

ABS

`ABS` returns the **absolute value** of a number. In other words, `ABS` converts a negative number to a positive number (if you pass a *positive* number, `ABS` does nothing).

```
PRINT ABS(12)
PRINT ABS(-12)
```

Output:

```
12
12
```

COS, SIN, TAN, and ATN

You can do the following trigonometric functions in QBasic:

```
COS   (Cosine)
SIN   (Sine)
TAN   (Tangent)
ATN   (Arctangent, inverse of TAN)
```

Example:

```
CONST PI = 3.141593

PRINT COS(PI / 4)
PRINT SIN(PI / 3)
PRINT TAN(-PI / 2)
PRINT ATN(TAN(-PI / 2))
```

Output:

```
.7071067
.8660254
6137956
1.570796  (Same as PI / 2)
```

Getting the current date and time

To get the current time, use the `TIME$` command.

```
PRINT TIME$
```

The above example returns **"military"** time. See the following figure:

```
hour 00 = 12 a.m. (midnight)
hours 01-11 = a.m.
hours 12-23 = p.m.
```

You can also use `TIME$` to *set* the time.

```
TIME$ = "15:30:00"  ' Sets current time to 3:30 p.m.
```

Date

To find out the current **date**, use the `DATE$` function.

```
PRINT DATE$
```

Like `TIME$`, you can also set the *date*.

```
DATE$ = "01/01/2000"
```

TIMER

Use `TIMER` to get the number of seconds since midnight.

```
PRINT TIMER
```

Output:

```
43199.99  (Just before noon)
```

Arrays

An array is a list of variables of the same type. Arrays are useful for organizing multiple variables. To create an array, use the `DIM` (dimension) command.

The following example does *not* use arrays:

```
a = 2
b = 4
c = 6
d = 8
e = 10

PRINT a, b, c, d, e
```

Output:

```
2          4          6          8          10
```

This uses an array called `vars`, which contains 5 variables:

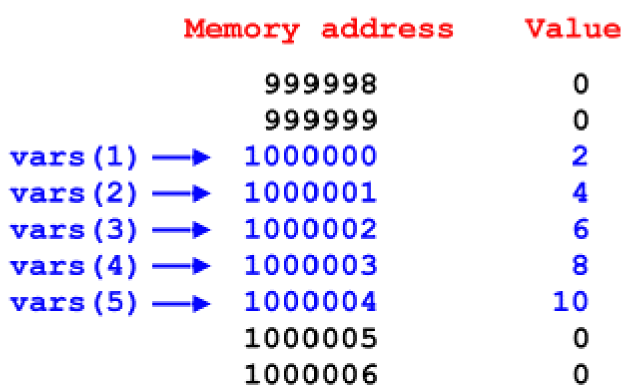
```
DIM vars(5)

' Each of these are separate variables:
vars(1) = 2
vars(2) = 4
vars(3) = 6
vars(4) = 8
vars(5) = 10

PRINT vars(1), vars(2), vars(3), vars(4), vars(5)
```

Output:

```
2          4          6          8          10
```



How an array of variables is stored in memory
(**NOTE:** Memory addresses are not necessarily as specified)

The above program can also be written like this:

```
DIM vars(5)

FOR x = 1 to 5
   vars(x) = x * 2
NEXT

FOR x = 1 to 5
   PRINT vars(x),
NEXT
```

Output:

```
2         4         6         8         10
```

Strings

You can also create an array of **string** variables.

```
DIM vars$(5)

vars$(1) = "Two"
vars$(2) = "Four"
vars$(3) = "Six"
vars$(4) = "Eight"
vars$(5) = "Ten"

PRINT vars$(1), vars$(2), vars$(3), vars$(4), vars$(5)
```

Output:

```
Two          Four         Six          Eight        Ten
```

Variable types

The non-string variables we've used in this tutorial are actually called **single-precision** variables. These types of variables (`SINGLE`'s) are used to store numbers that can contain a **decimal** value (such as **1.89** or **3.141593**). Since they have decimal values, they are also known as **"floating-point"** variables.

This chapter describes other types of variables used in QBasic.

```
INTEGER - A non-floating-point variable (no decimal value) that can
store integers between -32,768 and 32,767

LONG - Same as INTEGER, but can contain numbers between
-2,147,483,648 and 2,147,483,647.

DOUBLE - Same as SINGLE, but can have twice as many digits.
```

To define a variable's type, use `DIM` with the `AS` attribute.

```
DIM var1 AS INTEGER
DIM var2 AS LONG
DIM var3 AS DOUBLE

var1 = 15.28
var2 = -2000000000
var3 = 12345678.12345678

PRINT var1
PRINT var2
PRINT var3
```

Output:

```
15  (Notice how the decimal value is removed)
-2000000000
12345678.12345678
```

Using special characters

You can use special *characters* to specify a variable's type. These characters can also be used to specify a *number's* type.

To do so, place one of the following at the end of a variable (or number):

```
!   (single--actually, this doesn't change anything)

%   (integer)

&   (long)

#   (double)
```

```
$  (string--as we already know)
```

Example (notice the *number sign* on **12345678.12345678**):

```
var1% = 15.28
var2& = -2000000000
var3# = 12345678.12345678#

PRINT var1%
PRINT var2&
PRINT var3#
```

Output:

```
15
-2000000000
12345678.12345678
```

Subroutines and functions

A subroutine (also called a "module") is a **"mini-program"** inside your program. In other words, it is a collection of commands--and can be executed anywhere in your program.

To create a subroutine:

1. Go to the "Edit" menu
2. Select "New Sub"
3. Enter a name for the subroutine
4. Type a list of commands between SUB and END SUB

To use the subroutine:

1. Press F2
2. Select "Untitled"
3. Press Enter to return to the "main module"
4. Use CALL to execute the subroutine

> **TIP:** Another way to create a subroutine is by typing `SUB <name>` in the main module.
>
> **SUB MySub**

The following example does **not** use subroutines:

```
PRINT "Enter some text:";
INPUT text$
PRINT "The text you entered was: "; text$

PRINT "Enter some text:";
INPUT text$
PRINT "The text you entered was: "; text$

PRINT "Enter some text:";
INPUT text$
PRINT "The text you entered was: "; text$

PRINT "Enter some text:";
INPUT text$
PRINT "The text you entered was: "; text$

PRINT "Enter some text:";
INPUT text$
PRINT "The text you entered was: "; text$
```

```
PRINT "Enter some text:";
INPUT text$
PRINT "The text you entered was: "; text$

PRINT "Enter some text:";
INPUT text$
PRINT "The text you entered was: "; text$
```

By using a subroutine, the above program can be simplified like this:

```
CALL GetText
CALL GetText
CALL GetText
CALL GetText
CALL GetText
CALL GetText
CALL GetText


SUB GetText

   PRINT "Enter some text:";
   INPUT text$
   PRINT "The text you entered was: "; text$

END SUB
```

The following is even more concise:

```
FOR x = 1 TO 7
   CALL GetText
NEXT


SUB GetText
   PRINT "Enter some text:";
   INPUT text$
   PRINT "The text you entered was: "; text$
END SUB
```

Parameters

Parameters are numbers and strings that you pass to a subroutine, much like a QBasic command.

```
' This passes 16 as a parameter:

CALL OutputNumber(16)


' Notice the parentheses around the parameter "num."
' Any variables placed inside the parentheses are set as
```

```
' the subroutine's parameters.

SUB OutputNumber (num)

   PRINT num

END SUB
```

Output:

```
16
```

> **TIP:** Variables created in your program cannot be used in the subroutines unless you use `COMMON SHARED` (followed by a variable) in the main module.
>
> **COMMON SHARED x$**

Functions

A function is the same as a subroutine, except it **returns a value**. Also, you must leave out the `CALL` command.

To return a value, set a variable with the **same name** as the function.

```
PRINT Add(10, 7)


FUNCTION Add (num1, num2)

   Add = num1 + num2

END FUNCTION
```

Output:

```
17
```

Since a function can return a value, the name of the function can end with special characters (see *Variable types,* Using special characters).

```
' Notice the dollar sign ($) after "Add." It means
' the function returns a string.

PRINT Add$("Hello", "World")


FUNCTION Add$ (str1$, str2$)

   Add$ = str1$ + str2$
```

```
END FUNCTION
```

Output:

```
HelloWorld
```

Numbering systems

(This chapter is provided to help you understand certain parts of chapter 19, *Memory*.)

Normally, when we use a number such as **110**, we understand it to mean **"one hundred and ten,"** but in this chapter you will see how this is not always the case.

Hexadecimal numbers

We generally use the **base 10** (decimal) numbering system, where each digit must be between 0-9; but the "hexadecimal" system **(base 16)** can also have digits **A**, **B**, **C**, **D**, **E**, and **F** (16 total digits).

(The hexadecimal numbers in this tutorial are **red**.)

```
0 = Zero
1 = One
2 = Two
3 = Three
4 = Four
5 = Five
6 = Six
7 = Seven
8 = Eight
9 = Nine
A = Ten
B = Eleven
C = Twelve
D = Thirteen
E = Fourteen
F = Fifteen
```

In the base 10 system, you add another digit when you get past the number **9**; but with base 16, it isn't added until after **F** (or fifteen).

```
10 = Sixteen
11 = Seventeen
12 = Eighteen
13 = Nineteen
14 = Twenty
15 = Twenty one
16 = Twenty two
17 = Twenty three
18 = Twenty four
19 = Twenty five
1A = Twenty six
1B = Twenty seven
1C = Twenty eight
1D = Twenty nine
1E = Thirty
1F = Thirty one
20 = Thirty two
```

```
21 = Thirty three
22 = Thirty four
23 = Thirty five
24 = Thirty six
 .
 .
 .
```

In the decimal system (base 10), we multiply **ten** for each time a digit goes to the left.

```
   10 = 10
  100 = 10 * 10
 1000 = 10 * 10 * 10
10000 = 10 * 10 * 10 * 10
    .
    .
    .
```

But in the hexadecimal (base 16) system, we multiply **sixteen**, instead.

```
   10 = 16                 (16)
  100 = 16 * 16            (256)
 1000 = 16 * 16 * 16       (4096)
10000 = 16 * 16 * 16 * 16  (65536)
    .
    .
    .
```

Therefore, since 10 is 16 and 100 is 256, the number 110 is **two hundred and seventy two** (272).

```
110 = (100 + 10) = (256 + 16) = 272
```

(To download a number converter, click here.)

TIP: To enter a hexadecimal number in QBasic, use `&H`.

&H110

Binary numbers

The "binary" system **(base 2)** can only have two digits, **0** and **1**. Therefore, no binary number has a digit between 2 and 9.

(Binary numbers are shown in **dark blue**.)

```
    0 = Zero
    1 = One
   10 = Two
   11 = Three
  100 = Four
  101 = Five
  110 = Six
  111 = Seven
 1000 = Eight
 1001 = Nine
 1010 = Ten
 1011 = Eleven
 1100 = Twelve
 1101 = Thirteen
 1110 = Fourteen
 1111 = Fifteen
10000 = Sixteen
10001 = Seventeen
10010 = Eighteen
10011 = Nineteen
10100 = Twenty
    .
    .
    .
```

Notice how binary numbers can be found by excluding numbers that have a 2, 3, 4, 5, 6, 7, 8, or 9.

```
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
16
17
18
19
20
21
22
 .
 .
 .
```

```
 97
 98
 99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
  .
  .
  .
```

In base 10, as explained above, we multiply **ten** for each time a digit goes to the left.

```
   10 = 10
  100 = 10 * 10
 1000 = 10 * 10 * 10
10000 = 10 * 10 * 10 * 10
    .
    .
    .
```

But in binary, we multiply by **two**.

```
   10 = 2               (2)
  100 = 2 * 2           (4)
 1000 = 2 * 2 * 2       (8)
10000 = 2 * 2 * 2 * 2   (16)
    .
    .
    .
```

So, since 10 is 2 and 100 is 4, the number 110 is **six**.

```
110 = (10 + 100) = (2 + 4) = 6
```

(To download a number converter, click here.)

TIP: Binary (and hexadecimal) numbers are often written with leading **0**'s.

0000 (same as 0)
0001 (same as 1)
0010 (same as 10)
0011 (same as 11)

Memory

(Before you study this chapter, you may need to read chapter 18, *Numbering systems*.)

Bits

A "bit" is the smallest piece of data stored in your computer's memory. The value of a bit can be either **0** or **1**. All data in your computer has a certain number of bits.

Bytes

A "byte" is **8 bits**, and can have a value between 0 and 255 (or, in binary, between **0** and **11111111**). A character, such as `Q`, takes up one byte of memory. This is because there are 256 different characters.

(If you don't fully understand bits and bytes, don't worry about it.)

How data is stored

Data is stored in RAM at a certain **memory address**, as explained in chapter 3 (*Variables*). Each address takes up **1 byte** of memory. Therefore, it can only have a value between 0 and 255.

A memory address (on a 32-bit computer) can be somewhere between **0** and **4,294,967,295**. In hexadecimal, this is between **0** and **FFFFFFFF**.

Each memory address is divided into two parts: **segments** and **offsets**. See the figure below.

Segment	Offset			
0000	0000	=	Memory address:	00000000
0000	0001	=	Memory address:	00000001
0000	0002	=	Memory address:	00000002
0000	0003	=	Memory address:	00000003
0000	0004	=	Memory address:	00000004
⋮	⋮			⋮
FFFF	FFFB	=	Memory address:	FFFFFFFB
FFFF	FFFC	=	Memory address:	FFFFFFFC
FFFF	FFFD	=	Memory address:	FFFFFFFD
FFFF	FFFE	=	Memory address:	FFFFFFFE
FFFF	FFFF	=	Memory address:	FFFFFFFF

Segments and offsets

A memory address such as **12345678** (in hexadecimal) has a segment of **1234** and an offset of **5678**.

A segment can have a value between **0** and **65535** (or between **0** and **FFFF**). An offset can be within the same range.

You can find out a memory address of a piece of data by multiplying its segment by **65536** (or **10000**, in hexadecimal) and then adding its **offset** to the result. In QBasic, you can get a variable's segment by using `VARSEG` and its offset by using `VARPTR`.

```
segment = VARSEG(x)
offset = VARPTR(x)

' This prints the memory address of "x" (in decimal):

PRINT (segment * 65536) + offset
```