# C64 Workbook: From Hardware to Pong

Dr. Yves J. Hilpisch[1]

December 11, 2025

[1]Get in touch: `https://linktr.ee/dyjh`. Web page `https://hilpisch.com`. Research, structuring, drafting, and visualizations were assisted by GPT 5.1 as a co-writing tool under human direction.

# Contents


**1 How to Use This Workbook** **2**

**2 Getting Started with the VICE C64 Emulator** **2**
2.1 Installing and Launching VICE (Very Short Version) . . . 2
2.2 Basic Controls and First Program . . . 3

**3 C64 Hardware Overview** **4**
3.1 CPU, RAM, and ROM . . . 4
3.2 Video, Sound, and I/O Chips . . . 5
3.3 Memory Map Sketch . . . 5

**4 BASIC on the C64** **6**
4.1 The BASIC Environment . . . 6
4.2 Core BASIC Constructs . . . 7
4.3 Using PEEK and POKE . . . 7

**5 Assembly on the C64** **8**
5.1 Why Assembly? . . . 8
5.2 Assembly Tooling: Monitors, Cross-Assemblers, and IDEs . . . 9
5.3 A Tiny Assembly Program . . . 9

**6 Designing a Simple Pong Variant** **10**
6.1 Game Rules and Constraints . . . 10
6.2 Screen Layout and Coordinate System . . . 11

**7 Implementing Pong Step by Step** **12**
7.1 Initial Setup: Clearing the Screen and Drawing the Field . . . 12
7.2 Representing and Updating the Ball . . . 13
7.3 Handling Input for the Paddle . . . 14
7.4 Game Loop and Timing . . . 15
7.5 Basic Scoring and Restart Logic . . . 15

**8 Next Experiments and Extensions** **17**
8.1 Adding Sound Effects . . . 17
8.2 Polishing Graphics . . . 18
8.3 Beyond Pong . . . 19

**A Full Pong Listing and How to Run It** **19**
A.1 Full BASIC Listing . . . 19
A.2 Running the Game in VICE . . . 22
A.3 Transferring the Program from macOS to VICE . . . 22

# 1 How to Use This Workbook

This workbook is a hands-on companion to the C64 chapters in the main book [1]. It assumes you have access to a Commodore 64 emulator such as VICE on a modern machine (for example a Mac) and that you are willing to tinker, type code, and run experiments. You do not need prior experience with 8-bit hardware, but you should be comfortable typing short programs and restarting things when they do not work the first time.

The recommended way to use the material is:

- skim a section to see what you are about to build,
- implement and run the examples on your own emulator,
- and only then move on to the next step.

In other words, treat the text as a guide rather than as something to read passively. Small, working programs are worth much more than detailed plans that never get typed in.

The compiled PDF of this workbook is available at `https://hilpisch.com/commodore.pdf`.

**What You Will Build**

By the end of this workbook you will have:

- explored the basic hardware layout of the C64 (CPU, memory, ROM, I/O),
- written and run small BASIC and assembly programs,
- and implemented a simple Pong-style game step by step, including graphics, input, and basic scoring.

Code listings and screenshots are sketched as placeholders; you are encouraged to fill in the details and adapt them as you go.

# 2 Getting Started with the VICE C64 Emulator

This workbook pairs with a separate, more detailed VICE emulator manual, *VICE C64 Emulator for macOS: Installation and User Manual* [2]. Here we summarise just enough to get you from zero to a working virtual C64. When you want to dive deeper—for example into advanced controller mappings, drive emulation, or the built-in monitor—consult the full manual alongside this workbook.

## 2.1 Installing and Launching VICE (Very Short Version)

The goal of this subsection is simply to get the C64 screen with a `READY.` prompt running on your Mac; fine-tuning can wait.

- Download the appropriate macOS build of VICE from `https://vice-emu.sourceforge.io` (GTK3, Intel or Apple Silicon).
- Drag the VICE folder into *Applications*, then use macOS *Privacy & Security* settings to allow the app to run the first time.
- Open the VICE `bin` folder and launch `x64sc`, the cycle-accurate C64 emulator.
- Once the emulator window appears, you should see a blue screen with a BASIC startup message and the `READY.` prompt.

For a visual reminder of what that screen looks like in the emulator, Figure 1 shows a typical VICE window with the familiar blue background and `READY.` prompt.

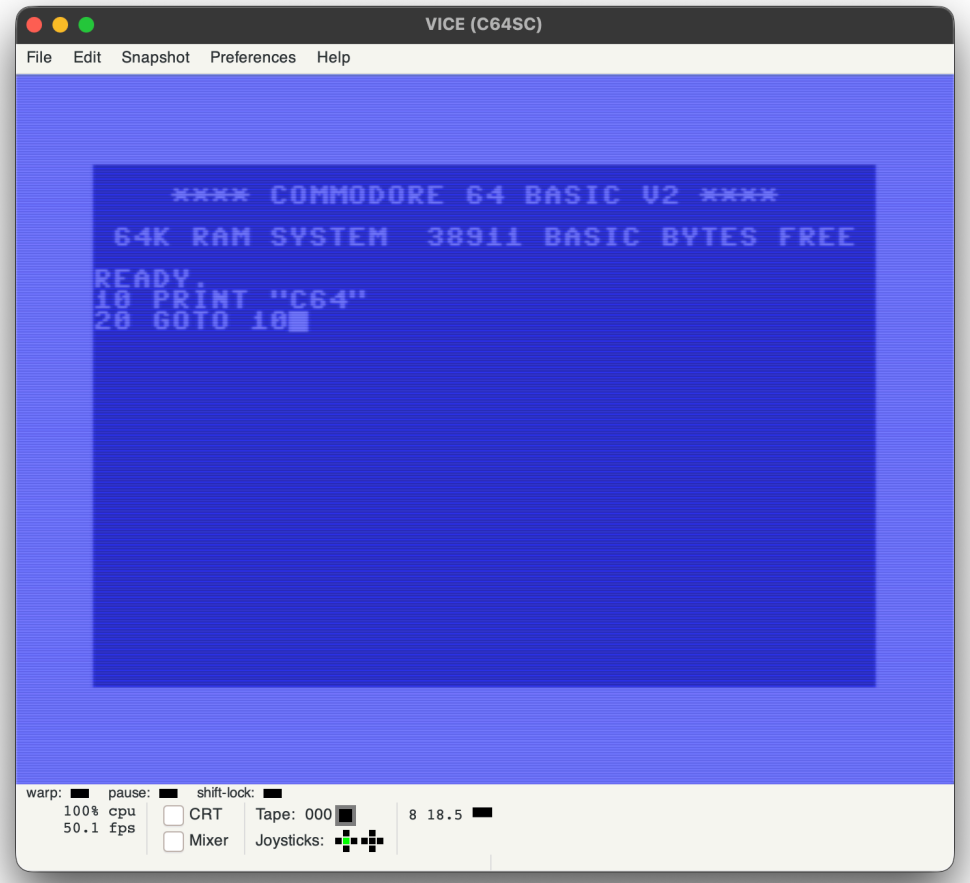


Figure 1: The VICE C64 emulator showing the classic blue startup screen with the BASIC banner and `READY.` prompt.

## 2.2 Basic Controls and First Program

Before we touch graphics or games, it is important to get comfortable entering, running, and stopping tiny programs.

- Use *Settings* → *Keyboard Settings* to pick *Symbolic* mapping so your Mac keyboard behaves as expected.
- At the `READY.` prompt, type:

  ```
  10 PRINT "HELLO, C64!" : GOTO 10
  RUN
  ```

- Stop the program with the emulator's *Stop* or *Reset* commands. Type

  ```
  LIST
  ```

Table 1: Key characteristics of the 6502/6510 CPU family.

| Feature | Approximate specification |
|---|---|
| Data bus width | 8 bits |
| Address bus width | 16 bits (up to 64 KB address space) |
| Main registers | Accumulator A; index registers X, Y (all 8 bit) |
| Stack pointer | 8 bit, stack in page `$0100`–`$01FF` |
| Status flags | Negative, Overflow, Break, Decimal, Interrupt, Zero, Carry |
| Typical C64 clock | $\approx 1$ MHz (PAL slightly below, NTSC slightly above) |
| Instruction set size | $\approx 56$ documented instructions, multiple addressing modes |
| Endianness | Little-endian (low byte at lower address) |

to see the stored program line again, then edit it and rerun with `RUN`.

- Experiment by changing the message, adding spaces, or lowering the scroll speed (for example by inserting a simple delay loop in BASIC).

**For Full Details**

This section stays intentionally short. For:

- full installation instructions,
- keyboard and joystick configuration,
- loading disk, tape, and program images,
- warp mode, snapshots, and the machine-language monitor,

consult the dedicated VICE manual [2]. The rest of this workbook assumes you can start `x64sc`, type BASIC, and load simple programs.

# 3 C64 Hardware Overview

This section zooms in on the C64 as a physical and logical machine. You will build a mental map of its processor, memory, graphics and sound chips, and the way all of these pieces are laid out in the 64 KB address space. Later, when you write BASIC and assembly code, you will keep returning to this picture to understand what each instruction really does inside the machine.

## 3.1 CPU, RAM, and ROM

To write meaningful programs and games, it helps to have a rough mental picture of what lives inside the C64: its processor, main memory, and built-in ROMs. At the heart of the machine sits a 6510 CPU, a close relative of the 6502 covered in the main history book. It is an 8-bit processor running at roughly 1 MHz, with a small set of registers (accumulator, index registers, stack pointer, status flags) and a simple instruction set. You do not need to memorise every flag or addressing mode to use this workbook, but it helps to remember that the CPU essentially performs a very fast, repeated cycle of "fetch an instruction from memory, decode it, execute it, and move on."

Table 1 summarises the key technical characteristics of the 6502/6510 family so you have a single place to check basic numbers while reading the rest of this section and the later assembly examples.

From the programmer's point of view, the C64 offers 64 KB of addressable memory. Internally, that 64 KB space is shared between:

- **RAM** (read/write memory) where your programs and data live during execution,
- **ROM** (read-only memory) containing the BASIC interpreter and KERNAL system routines,
- and **memory-mapped I/O** regions used to talk to chips like the VIC-II graphics processor and SID sound generator.

The BASIC interpreter itself lives in ROM, as does the KERNAL code that handles low-level tasks such as keyboard input and screen output. When you type a BASIC line and press `RETURN`, the CPU fetches code from the BASIC ROM to parse and execute your command; when you call routines such as `PRINT` or `OPEN`, KERNAL code in ROM manages the details. Understanding that some addresses correspond to RAM you control and others correspond to permanent firmware is the first step toward using `PEEK` and `POKE` effectively later in this workbook.

## 3.2 Video, Sound, and I/O Chips

Beyond the core processor and memory, dedicated chips handle graphics, sound, and input/output. The VIC-II graphics chip is responsible for what you see on the screen. In its standard text mode it treats the display as a grid of 40 columns by 25 rows of characters, each character drawn from a built-in font. It also supports hardware sprites—small movable objects that can fly over the background without you having to redraw the whole screen—and a palette of colours that can be applied to characters, borders, and sprites. Much of simple game programming on the C64 revolves around learning which memory locations the VIC-II watches for character data, colours, and sprite positions.

The SID (Sound Interface Device) chip plays the role of a tiny synthesiser. It provides three independent voices, each with selectable waveforms (pulse, sawtooth, triangle, noise), adjustable envelopes, and simple filtering. By writing values into a handful of SID registers, you can create everything from short beeps for Pong-style collisions to more elaborate music. Early in this workbook we will stick to simple effects; later you can return to these concepts to add more expressive sound.

Input and general-purpose I/O are managed by additional interface adapters. They scan the keyboard matrix so that key presses can be read by the CPU, handle joystick ports, and talk to external devices such as disk drives over the serial bus. Even if you never learn the full details of these chips, it is useful to know that pressing a key, moving a joystick, or loading a file all boil down to changes in specific memory-mapped registers that the CPU can read.

## 3.3 Memory Map Sketch

The C64's memory map ties these pieces together by assigning ranges of addresses to RAM, ROM, screen memory, and I/O registers. At a high level, every address from 0 to 65535 belongs to exactly one of these categories. Some especially important regions are:

- **Zero page** (`$0000`–`$00FF`): a small block of RAM with special fast-addressing modes, often used for frequently accessed variables and pointers.
- **Stack** (`$0100`–`$01FF`): the area the CPU uses to store return addresses and temporary values during subroutine calls.
- **Screen RAM** (by default around `$0400` in RAM): one byte per character position in the 40×25 text grid.
- **Colour RAM** (in a dedicated block near `$D800`): one nibble per character position that controls its foreground colour.

- **I/O and character ROM region** (`$D000`–`$DFFF`): home to VIC-II, SID, and other registers, with the character set ROM available behind them when configured.
- **BASIC ROM** (`$A000`–`$BFFF`) and **KERNAL ROM** (`$E000`–`$FFFF`): firmware that implements BASIC and core system services.

You will not need to memorise every hex range, but you will often need to know "roughly where" something lives. Later sections will refer back to this sketch when we `POKE` values into screen memory, change border colours, or read joystick input. If you enjoy visual overviews, sketch a simple diagram of the address space in your notes, with labelled blocks for RAM, ROM, and I/O; you will find yourself referring to it frequently.

# 4 BASIC on the C64

With the hardware picture in mind, this section focuses on the language you meet first when the C64 boots: BASIC. You will learn how programs are stored and edited, how variables and control flow work, and how to use `PEEK` and `POKE` to reach through BASIC into the underlying memory map. These skills form the bridge from simple text programs to interactive games.

## 4.1 The BASIC Environment

The built-in BASIC interpreter is your first programming environment on the C64, providing a prompt, line editor, and simple commands for saving and loading code. When you see the `READY.` prompt, the machine is already waiting for BASIC commands. Anything you type without a line number executes immediately; anything you type *with* a line number is stored as part of a program. For example:

```
10 PRINT "C64 WORKBOOK"
20 PRINT "HELLO AGAIN"
RUN
```

adds two lines to your current program and then runs it. You can inspect what is stored in memory with:

```
LIST
```

and delete or change individual lines by retyping them. Typing `NEW` clears the current program; be careful with it, as there is no undo.

To save and load programs, you use BASIC commands that talk to whatever storage device VICE emulates. On a virtual disk in drive 8, a typical sequence might be:

```
SAVE "PONG",8
NEW
LOAD "PONG",8
RUN
```

In practice, the exact device number and filenames depend on how you attach disk images in VICE, but the pattern—`SAVE`, `LOAD`, `RUN`—remains the same. It is worth practising these commands a few times until you can type them almost without thinking.

C64 BASIC is intentionally simple. It lacks modern structured control flow (there are no `WHILE` or `REPEAT` keywords), has limited built-in graphics and sound commands, and treats most numbers as a single numeric type. Those limitations are part of why this workbook leans so heavily on direct memory access: you will often use BASIC as a thin layer over the underlying hardware, calling into ROM routines or manipulating memory-mapped registers yourself.

## 4.2 Core BASIC Constructs

Before building larger programs, you will use a small set of core constructs repeatedly. Variables in C64 BASIC are created simply by using them. The first time you write `X=10`, the interpreter allocates space for `X` and stores the value. Variable names may contain letters and (in some cases) digits, but only the first two characters are usually significant, so `SCORE` and `SC` refer to the same underlying variable. This is a quirk worth remembering when you choose names.

Expressions combine variables, numbers, and operators. For example:

```
10 A = 5
20 B = 3
30 C = A + B * 2
40 PRINT C
RUN
```

prints `11`. BASIC follows the usual precedence rules (multiplication before addition), but when in doubt you can add parentheses.

Control flow is built from `IF...THEN`, `GOTO`, and `FOR...NEXT`. A simple branching example is:

```
10 INPUT "GUESS A NUMBER 1-5 "; G
20 IF G = 3 THEN PRINT "CORRECT!" : GOTO 40
30 PRINT "TRY AGAIN." : GOTO 10
40 PRINT "GAME OVER"
RUN
```

Loops can be written explicitly with `GOTO` or more compactly with `FOR...NEXT`:

```
10 FOR I = 1 TO 10
20 PRINT "I ="; I
30 NEXT I
RUN
```

Input and output use `INPUT` and `PRINT`. The semicolon in `PRINT "I ="; I` suppresses the automatic newline between arguments, giving you more control over how text appears. As you work toward Pong, you will use these constructs to keep track of positions, update scores, and respond to player input.

## 4.3 Using PEEK and POKE

The `PEEK` and `POKE` commands let BASIC reach beneath its usual abstractions and touch individual memory locations directly. At a conceptual level, `PEEK(A)` asks "what byte currently lives at address `A`?" and `POKE A,B` says "store the value `B` at address `A`." The addresses are written as decimal numbers in BASIC, even though many reference books list them in hexadecimal.

One classic experiment is to change the screen border and background colours. On the C64, the border colour register lives at address `53280` and the main background colour at `53281`. Try:

```
10 POKE 53280,0   : REM BORDER BLACK
20 POKE 53281,6   : REM BACKGROUND BLUE
30 GOTO 30
RUN
```

The program sets the colours once and then loops forever. You can interrupt it and experiment with different numbers; small integers between 0 and 15 select different colours. To see what the current border colour is, type:

```
PRINT PEEK(53280)
```

and BASIC will print the stored value.

Another useful experiment is to write directly into screen memory. By default, the top-left character cell of the text screen corresponds to a location in RAM that you can `POKE`. The exact address depends on how the VIC-II is configured, but a typical starting point is:

```
10 POKE 1024,1
20 GOTO 20
RUN
```

This stores the value `1` at the first screen position, which the character generator interprets as a particular glyph. Changing the value changes the character; moving to nearby addresses changes neighbouring cells. Even without memorising all the codes, you will feel the direct connection between numeric values in memory and symbols on the screen.

These tiny programs hint at why `PEEK` and `POKE` are so powerful. They let you use BASIC as a readable control script while still taking advantage of the C64's specialised hardware. Later chapters in this workbook will extend the same idea to sprite positions, joystick input, and simple sound effects, and will eventually show how assembly language offers an even more precise way to work with the same memory map.

# 5 Assembly on the C64

Once you are comfortable with BASIC and direct memory access, the next step is to work in the language the CPU executes most directly: assembly. This section explains why assembly remains useful on an 8-bit machine, introduces common tools for writing it, and walks through a first tiny program so you can see how symbolic instructions turn into visible effects on the screen.

## 5.1 Why Assembly?

Assembly language sits one step above raw machine code and gives you fine-grained control over what the CPU does each cycle. On a modern machine, you can get far without ever touching assembly. On an 8-bit system with tight memory and timing budgets, being able to reason at this level often makes the difference between "nice demo" and "responsive game."

There are three main reasons to learn a little assembly for the C64:

1. **Speed.** BASIC interprets your program line by line at runtime. Even simple loops can become sluggish, especially if they manipulate the screen or perform arithmetic inside tight timing windows. Assembly code, once assembled, runs at full CPU speed.

2. **Precision.** Assembly lets you decide exactly which registers to use, which memory locations to touch, and when to enable or disable interrupts. That level of control is invaluable for flicker-free animation, stable scrolling, or music playback.

3. **Access to features.** Some hardware capabilities are awkward or impossible to reach comfortably from BASIC alone. Sprite multiplexing, cycle-exact raster effects, and efficient collision detection all benefit from assembly routines.

Historically, many commercial C64 games used a hybrid approach: BASIC for menus, high scores, and simple glue logic; assembly for the inner loops that handled graphics, sound, and game logic. This workbook follows the same spirit. You can get a basic Pong implementation working in fast, well-structured BASIC, and then selectively move performance-critical pieces into assembly as your confidence grows.

## 5.2 Assembly Tooling: Monitors, Cross-Assemblers, and IDEs

To write assembly you need two things: a way to turn human-readable instructions into machine code, and a way to load and run that machine code on the C64 or in the emulator. There are several workable paths; you can pick the one that fits your habits best.

The most direct tool is the built-in machine-code monitor that comes with the VICE emulator. From the VICE menu you can open the monitor and then:

- inspect registers and memory at specific addresses,
- step through instructions one by one,
- and enter small programs directly as hexadecimal opcodes.

This feels close to the metal and is excellent for understanding what the CPU really does, but it quickly becomes tedious for anything beyond a few instructions.

A more comfortable approach is to use a *cross-assembler* on your host machine. Tools such as ACME or KickAssembler let you write 6502/6510 assembly in a text editor, with labels and comments, and then assemble the code into a `.prg` file that can be loaded in VICE like any other program. The typical workflow looks like this:

1. Write an assembly source file with a clear starting address (`* = $C000`, for example) and labelled routines.
2. Run the assembler on your Mac to produce a binary `.prg`.
3. Attach the resulting file in VICE and `SYS` to the starting address from BASIC to jump into your code.

Modern integrated development environments (IDEs) wrap these steps with keyboard shortcuts and live rebuilds, but the core idea is the same: edit, assemble, load, run, inspect.

For the exercises in this workbook you can start with whichever method feels less intimidating. If you enjoy seeing exactly which bytes go where, the VICE monitor is a good first stop. If you prefer editing text files and reusing your work, a cross-assembler quickly pays off.

## 5.3 A Tiny Assembly Program

As a bridge from BASIC to assembly, it is useful to start with a tiny program that does something familiar, such as writing a character on screen. Suppose you want to place the letter `A` in the top-left corner using assembly instead of `POKE`.

In pseudo-assembly, the program might look like this:

```
; Simple program: write 'A' to top-left screen cell
        * = $C000         ; load address for the program

START   LDA #$01          ; character code for 'A'
        STA $0400         ; store in first screen position
        RTS               ; return to BASIC
```

Here `LDA #$01` loads the accumulator register with the immediate value `$01`. The `#` symbol means "use this value directly," not "look up whatever lives at address `$0001`." The instruction `STA $0400` then stores the accumulator into the memory location corresponding to the first screen character cell. Finally, `RTS` (return from subroutine) hands control back to whoever called this code.

Once assembled to address `$C000`, you can load the program into VICE and call it from BASIC with:

since decimal 49152 is hexadecimal $C000. If everything is wired up correctly, the top-left character on the screen will change when you run the SYS command.

Even this tiny example illustrates several key ideas:

- Labels such as START make it possible to refer to locations symbolically rather than memorising raw addresses.
- Immediate values (#$01) and absolute addresses ($0400) represent two different addressing modes, both of which you will use constantly.
- BASIC and assembly can cooperate smoothly: BASIC sets things up and calls SYS, assembly performs fast low-level work, and RTS returns to BASIC when done.

In later sections you can extend this pattern to move characters around the screen, update sprite positions, or handle time-critical parts of a game loop.

# 6 Designing a Simple Pong Variant

Before you start typing the first line of game code, it helps to treat Pong as a small design project. This section turns an informal idea—"a paddle, a ball, and a score"—into concrete rules, a screen layout, and data structures. That design will guide the implementation sections that follow and make it easier to debug when something does not behave as expected. To keep the code and explanations focused on core ideas like the game loop and memory layout, the baseline in this workbook is a *single-player* variant: one paddle you control on the left, and a simple wall on the right that always reflects the ball. Two-paddle versions then become a natural extension once you understand the simpler case.

## 6.1 Game Rules and Constraints

Designing even a simple Pong variant benefits from writing down rules and constraints before touching code. You can think of this as the game's "specification" in plain language, something you can read to check whether your implementation matches your intentions.

A minimal single-player variant—the one implemented step by step in this workbook—might look like this:

- You control one vertical paddle on the left side of the screen using up/down keys or a joystick.
- A ball moves across the screen, bouncing off the top and bottom edges.
- When the ball reaches the left side, you either hit it with the paddle (and it bounces back) or miss it (and you lose a point or a life).
- The game keeps track of your score or number of successful returns.

The right-hand side of the court acts as a fixed wall: whenever the ball hits it, it simply bounces back. You do not control a second paddle in the baseline implementation, which keeps collision logic and input handling straightforward.

You can extend this design later to two paddles (player vs player), variable ball speeds, or multiple balls, but starting with a single paddle and a wall simplifies the logic and makes it easier to see how each line of code affects behaviour.

On the implementation side, you should decide which language or mix of languages to use. One sensible sequence is:

1. build the first complete, working version in BASIC, using `PEEK` and `POKE` for screen updates;
2. measure whether the responsiveness is good enough in VICE; and
3. if not, identify one or two inner loops (ball motion, collision detection) to rewrite in assembly.

The C64's hardware imposes constraints that shape your design whether you write BASIC or assembly:

- **Frame rate and timing.** At roughly 50 Hz (PAL) or 60 Hz (NTSC), you have a limited number of CPU cycles per visual frame. Expensive calculations inside the game loop can cause jitter or sluggish controls.
- **Screen resolution.** In character mode you effectively have a 40×25 grid. Paddles and balls that move in whole-character steps will look coarse but are much simpler to implement than pixel-perfect motion.
- **Input latency.** Reading joysticks or keys too infrequently makes the paddle feel unresponsive. Reading them every frame, in a predictable place in the loop, keeps controls crisp.
- **Memory limits.** A full 64 KB may sound generous, but graphics data, sound effects, and code all share it. Keeping your design compact early on leaves room for later experiments.

Writing these points down in your own words before coding creates a checklist you can revisit whenever you are unsure what to implement next.

## 6.2 Screen Layout and Coordinate System

Once you know the rules, you can sketch how the playfield will look on the C64's screen and where game objects will live in memory. A simple text-mode layout might reserve the full 40-character width and 25-character height for the court, with a border made of characters such as `"|"` and `"-"`.

One convenient coordinate system is:

- rows numbered from 0 at the top to 24 at the bottom,
- columns numbered from 0 on the left to 39 on the right,
- and a function or formula that maps a (row, column) pair to a specific address in screen memory.

In the default configuration, the top-left cell at row 0, column 0 corresponds to the base address of screen RAM (often `$0400`). Each row then occupies 40 consecutive bytes. In BASIC, you can capture this mapping with a small helper:

```
1000 REM MAP (ROW, COL) TO SCREEN ADDRESS
1010 DEF FNADDR(R,C) = 1024 + R*40 + C
```

assuming decimal `1024` as the base. Then drawing a paddle segment becomes:

```
POKE FNADDR(10,1), 81   : REM 'Q' AS PADDLE BLOCK
```

Using a function like this keeps your game logic expressed in terms of rows and columns rather than raw addresses, which makes it easier to reason about movement and collisions.

You also need to choose how to represent the paddle and ball visually. For a first version, treating both as single characters is enough:

- the paddle as a vertical stack of, say, three or four identical characters near the left edge;
- the ball as a single character such as `"*"` that moves one cell at a time.

Later, you can swap these characters for sprites controlled by the VIC-II, which allows smoother motion and frees the text screen for score displays and messages. The underlying coordinate logic—positions, velocities, collision checks—remains largely the same, making that upgrade a natural extension once the basic version works.

The overall arrangement of borders, paddle, ball, and score area is shown schematically in Figure 2. Keep that picture in mind as you read the code in the implementation section; each variable and `POKE` ultimately corresponds to one part of that court.

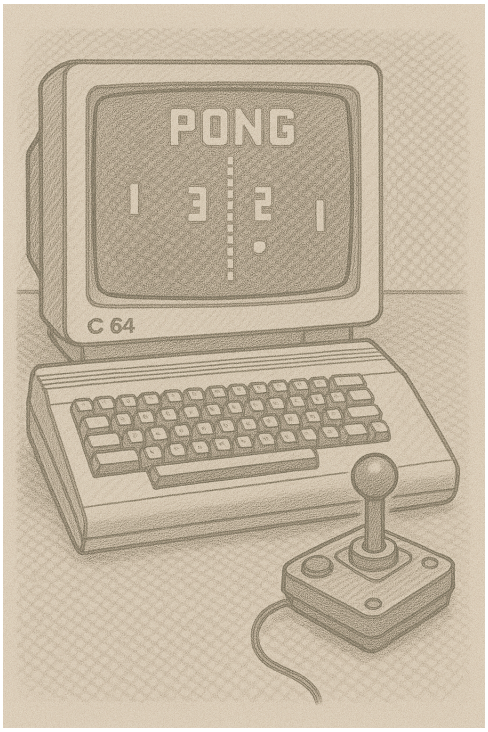


Figure 2: Schematic layout of a simple C64 Pong court. In the baseline single-player version implemented in this workbook, you control the left-hand paddle while the right-hand side acts as a fixed wall that reflects the ball. A true second paddle only appears in the suggested two-player extension.

# 7 Implementing Pong Step by Step

With the design in place, this section walks through a complete implementation of a simple Pong variant. You will start by clearing and drawing the playfield, then add a moving ball, responsive paddle controls, a timing loop, and basic scoring. At each stage, you can run the intermediate program in VICE, observe its behaviour, and decide whether to refine or extend it.

## 7.1 Initial Setup: Clearing the Screen and Drawing the Field

The first implementation step is to get a clean playfield on screen and verify that you can reliably draw static elements where you expect them. Start by defining a few constants and utility routines. The following code assumes the standard text screen lives at decimal address `1024` and reserves row 0 for the score display, with the playfield starting at row 1:

```
10 REM === CONSTANTS AND SETUP ===
20 SCR = 1024          : REM SCREEN BASE ADDRESS
30 WIDTH = 40          : REM COLUMNS
40 HEIGHT = 25         : REM ROWS (0..24)
50 TOPROW = 1          : REM TOP OF PLAYFIELD
60 BOTROW = 23         : REM BOTTOM OF PLAYFIELD
70 LEFTCOL = 1         : REM LEFT WALL COLUMN
80 RIGHTCOL = 38       : REM RIGHT WALL COLUMN
90 PADDLELEN = 4       : REM PADDLE HEIGHT IN CELLS
100 BALLCHAR = 81      : REM CHARACTER CODE FOR BALL
110 PADDLECHAR = 81    : REM CHARACTER CODE FOR PADDLE
```

You can adapt the exact character codes later; using the same code for paddle and ball keeps things simple at first.

Next, define the address-mapping helper from the previous section so that the rest of the program can work in row/column coordinates:

```
1000 REM MAP (ROW, COL) TO SCREEN ADDRESS
1010 DEF FNADDR(R,C) = SCR + R*WIDTH + C
```

Clearing the screen can be done either by printing the clear-screen character or by writing spaces to every cell. For clarity, use the built-in clear-screen code:

```
120 PRINT CHR$(147)    : REM CLEAR TEXT SCREEN
```

Now draw the static borders of the court and initialise paddle and ball positions:

```
130 REM === DRAW BORDERS ===
140 FOR C = LEFTCOL TO RIGHTCOL
150   POKE FNADDR(TOPROW, C), 45      : REM '-' TOP
160   POKE FNADDR(BOTROW, C), 45      : REM '-' BOTTOM
170 NEXT C
180 FOR R = TOPROW TO BOTROW
190   POKE FNADDR(R, LEFTCOL), 124    : REM '|' LEFT WALL
200   POKE FNADDR(R, RIGHTCOL), 124   : REM '|' RIGHT WALL
210 NEXT R
```

Finally, choose starting positions for the paddle and the ball:

```
220 REM === INITIAL POSITIONS ===
230 PADDLECOL = LEFTCOL + 1
240 PADDLETOP = INT((TOPROW+BOTROW)/2) - PADDLELEN/2
250 BALLR = INT((TOPROW+BOTROW)/2)
260 BALLC = INT((LEFTCOL+RIGHTCOL)/2)
270 DR = -1      : REM BALL VERTICAL STEP PER FRAME
280 DC = 1       : REM BALL HORIZONTAL STEP PER FRAME
```

At this point you have a cleared screen, a rectangular court, and variables that describe where the paddle and ball will live. You can test the address-mapping function by poking a few characters by hand before continuing.

## 7.2 Representing and Updating the Ball

With the field in place, you can focus on the ball: how to represent its position and motion and how to move it one frame at a time. The variables `BALLR` and `BALLC` hold the ball's current row and column; `DR` and `DC` hold its vertical and horizontal step per frame. A simple subroutine for drawing and erasing the ball looks like this:

```
2000 REM === DRAW BALL AT (BALLR, BALLC) ===
2010 POKE FNADDR(BALLR, BALLC), BALLCHAR
2020 RETURN

2030 REM === ERASE BALL AT (BALLR, BALLC) ===
2040 POKE FNADDR(BALLR, BALLC), 32      : REM SPACE
2050 RETURN
```

Each frame of the game you perform three actions in order:

1. erase the ball at its old position,
2. update the position based on `DR` and `DC` and handle bounces,
3. draw the ball at its new position.

The bounce logic for the top and bottom of the court and the right wall can be coded as:

```
2100 REM === UPDATE BALL POSITION AND HANDLE BOUNCES ===
2110 GOSUB 2030                 : REM ERASE OLD BALL
2120 BALLR = BALLR + DR
2130 BALLC = BALLC + DC

2140 REM BOUNCE OFF TOP AND BOTTOM
2150 IF BALLR <= TOPROW+1 THEN DR = 1 : BALLR = TOPROW+2
2160 IF BALLR >= BOTROW-1 THEN DR = -1 : BALLR = BOTROW-2

2170 REM BOUNCE OFF RIGHT WALL
2180 IF BALLC >= RIGHTCOL-1 THEN DC = -1 : BALLC = RIGHTCOL-2

2190 GOSUB 2000                 : REM DRAW NEW BALL
2200 RETURN
```

Here the ball never enters the border characters themselves; instead, the code checks one cell inside the top and bottom and adjusts direction as needed. The right wall acts as a solid reflector. The left side will be treated differently because it contains the paddle and the "miss" zone, which we handle together with paddle logic and scoring.

## 7.3 Handling Input for the Paddle

For interaction, this workbook treats the keyboard-as-joystick mapping in VICE as the default input method, with physical joysticks as an optional enhancement. A straightforward scheme is to use the `GET` command to read single-key input from the keyboard every frame. You can, for example, map `Q` to "move paddle up" and `A` to "move paddle down":

```
2300 REM === DRAW PADDLE BASED ON PADDLETOP ===
2310 FOR R = TOPROW+1 TO BOTROW-1
2320   POKE FNADDR(R, PADDLECOL), 32        : REM ERASE OLD PADDLE
2330 NEXT R
2340 FOR I = 0 TO PADDLELEN-1
2350   POKE FNADDR(PADDLETOP+I, PADDLECOL), PADDLECHAR
2360 NEXT I
2370 RETURN

2380 REM === HANDLE KEYBOARD INPUT FOR PADDLE ===
```

```
2390 GET K$ : IF K$ = "" THEN 2440
2400 IF K$ = "Q" AND PADDLETOP > TOPROW+1 THEN PADDLETOP = PADDLETOP-1
2410 IF K$ = "A" AND PADDLETOP+PADDLELEN < BOTROW THEN PADDLETOP = PADDLETOP+1
2420 GOSUB 2300
2430 REM IGNORE OTHER KEYS
2440 RETURN
```

This subroutine first erases the old paddle by writing spaces down the paddle column, then redraws the paddle at the new `PADDLETOP`. The input routine calls `GET K$` once per frame so that even brief key presses are captured reliably. Boundary checks ensure the paddle does not leave the playfield.

If you later prefer joystick input, you can replace the `GET` section with reads from the joystick port via the appropriate memory-mapped registers, leaving the drawing logic unchanged.

## 7.4 Game Loop and Timing

Everything comes together in the main loop, which coordinates input, state updates, drawing, and timing. A minimal loop that ties together paddle input and ball motion looks like this:

```
3000 REM === INITIAL DRAW OF PADDLE AND BALL ===
3010 GOSUB 2300            : REM DRAW PADDLE
3020 GOSUB 2000            : REM DRAW BALL

3100 REM === MAIN GAME LOOP ===
3110 GOSUB 2380            : REM HANDLE INPUT AND REDRAW PADDLE
3120 GOSUB 2100            : REM UPDATE BALL AND REDRAW
3130 FOR T = 1 TO 80 : NEXT T   : REM SIMPLE DELAY
3140 GOTO 3110
```

The delay loop on lines 3130–3140 is a crude but effective way to slow the game down so that it feels playable. You can experiment with different upper bounds for `T` depending on whether VICE is running in "warp" mode and how responsive you want the game to be. More advanced versions can synchronise with the vertical blank interval of the display using interrupts, but a busy-wait loop is sufficient for a first implementation.

The structure is intentionally simple: each frame consists of one pass through input handling and one pass through ball physics, followed by a short pause. Because all visual updates go through the same mapping function `FNADDR`, your mental model can stay focused on rows and columns.

## 7.5 Basic Scoring and Restart Logic

Adding a minimal scoring and restart system turns a moving ball and paddle into an actual game with goals and feedback. For scoring, you maintain two variables, `SCORE` and `LIVES`. A straightforward first version simply prints updates as text so you can see what is happening without yet worrying about a perfectly aligned status bar:

```
300 REM === SCORE AND LIVES ===
310 SCORE = 0
320 LIVES = 3
330 PRINT : PRINT "SCORE:"; SCORE; "  LIVES:"; LIVES
```

You can adjust the exact formatting to taste; the important part is that the status line does not interfere with the playfield rows.

Next, extend the ball-update routine to detect when the ball reaches the paddle column or slips past it. If the ball's column would move to the left of the paddle, you treat that as a miss and decrement `LIVES`; if the ball hits the paddle, you bounce it and increase `SCORE`:

```
2165 REM CHECK FOR PADDLE HIT OR MISS ON LEFT SIDE
2166 IF BALLC <= PADDLECOL THEN GOSUB 2600
```

and add a new subroutine starting at line 2600:

```
2600 REM === HANDLE COLLISION WITH LEFT SIDE ===
2610 REM IS THE BALL WITHIN THE PADDLE'S VERTICAL RANGE?
2620 IF BALLR >= PADDLETOP AND BALLR < PADDLETOP+PADDLELEN THEN GOSUB 2700
2625 IF BALLR >= PADDLETOP AND BALLR < PADDLETOP+PADDLELEN THEN RETURN
2630 REM OTHERWISE, THE PLAYER MISSES
2640 LIVES = LIVES - 1
2650 GOSUB 2800              : REM UPDATE STATUS LINE
2660 IF LIVES = 0 THEN GOSUB 2900 : END
2670 REM RESET BALL TO CENTRE AND CONTINUE
2680 BALLR = INT((TOPROW+BOTROW)/2)
2690 BALLC = INT((LEFTCOL+RIGHTCOL)/2)
2695 DR = -1 : DC = 1
2696 RETURN
```

When the ball does intersect the paddle, you reverse the horizontal direction and increment the score:

```
2700 REM === SUCCESSFUL PADDLE HIT ===
2710 DC = 1                      : REM SEND BALL BACK TO THE RIGHT
2720 SCORE = SCORE + 1
2730 GOSUB 2800                  : REM UPDATE STATUS LINE
2740 RETURN
```

The status update routine centralises how score and lives are printed:

```
2800 REM === PRINT SCORE AND LIVES ===
2810 PRINT "SCORE:"; SCORE; "  LIVES:"; LIVES
2820 RETURN
```

Finally, define a short game-over routine so that losing all lives feels like a clear event rather than an abrupt stop:

```
2900 REM === GAME OVER MESSAGE ===
2910 PRINT : PRINT "GAME OVER"
2920 PRINT "FINAL SCORE:"; SCORE
2930 RETURN
```

Putting these pieces together yields a basic but complete game:

- the main loop continually calls input and ball-update routines,
- the paddle responds to `Q`/`A`,
- the ball bounces off the top, bottom, and right walls,
- hitting the paddle increases the score, while missing costs a life and resets the ball.

Once you have this version running smoothly in VICE, you can experiment with faster ball speeds, varying bounce angles based on where the ball hits the paddle, or two-player variants that use both sides of the court.

# 8 Next Experiments and Extensions

Once you have a working Pong clone, you have a miniature playground for experimenting with the C64's capabilities. This final section suggests directions for extending the game with sound, improved graphics, and new mechanics so that the workbook becomes a launchpad for your own projects rather than an endpoint.

## 8.1 Adding Sound Effects

Simple sound effects can make even a minimal game feel more alive. The SID chip exposes several registers in memory that you can `POKE` to start and stop tones. For a first experiment, you do not need to understand every detail of envelopes and filters; it is enough to learn how to trigger a short beep when the paddle hits the ball.

On a standard C64, the first voice of the SID can be controlled via addresses around decimal `54272`. A minimal "hit" sound might look like this:

```
4000 REM === SIMPLE HIT SOUND ON SID VOICE 1 ===
4010 POKE 54276, 17      : REM SET ATTACK/DECAY
4020 POKE 54277, 240     : REM SET SUSTAIN/RELEASE
4030 POKE 54272, 34      : REM FREQUENCY LOW BYTE
4040 POKE 54273, 8       : REM FREQUENCY HIGH BYTE
4050 POKE 54278, 33      : REM ENABLE PULSE WAVEFORM AND GATE
4060 FOR T = 1 TO 100 : NEXT T
4070 POKE 54278, 32      : REM TURN GATE BIT OFF
4080 RETURN
```

The overall control flow of the program is sketched in Figure 3, which groups related lines into initialisation, main loop, and event-handling stages. Use it as a map when you read or debug the listing: every arrow corresponds to a `GOSUB` or `GOTO` in the code.

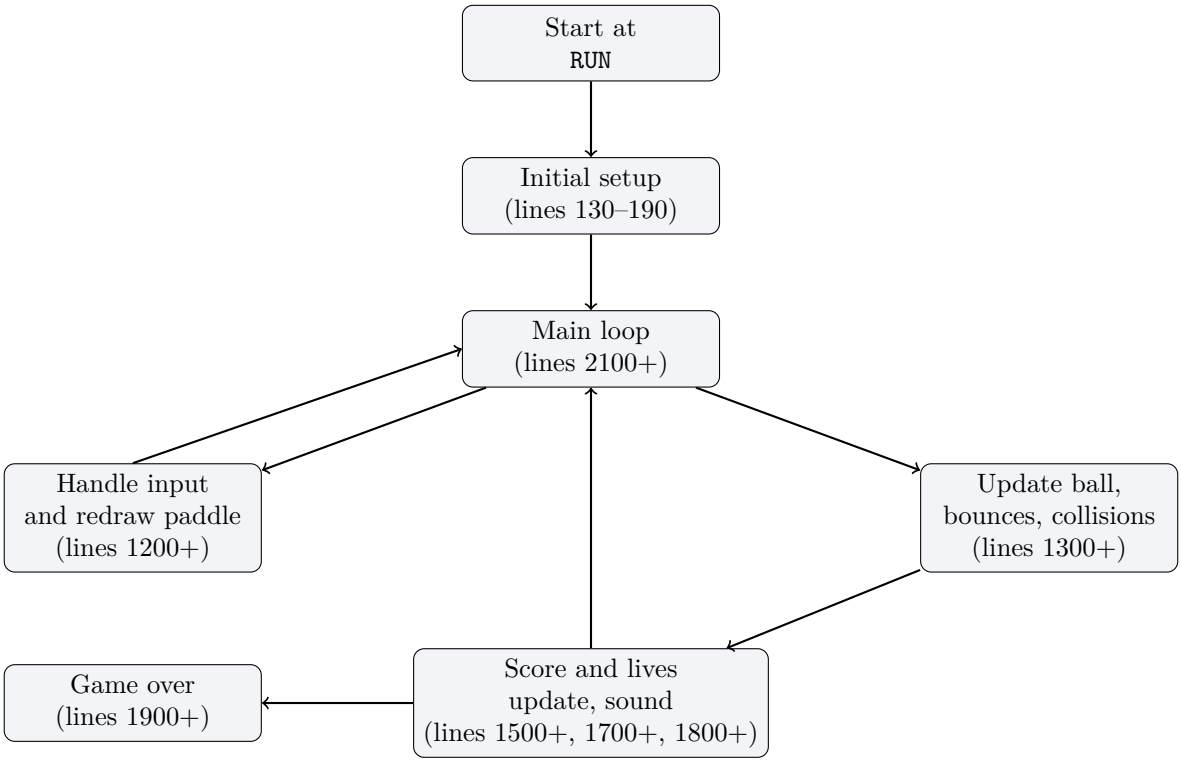


Figure 3: Schematic control-flow view of the C64 Pong program: initial setup leads into a loop that repeatedly handles input, updates the ball, and adjusts score and lives until a game-over condition is reached.

You can call `GOSUB 4000` from the successful-hit routine at line 2700 so that every paddle hit produces a short tone. Experiment with the frequency and envelope values to find a sound that feels satisfying without being annoying over many repetitions.

If you later want richer audio, you can dedicate one voice to hits, one to wall bounces, and one to a simple background tone, all controlled by small subroutines that you call from different parts of the game loop.

## 8.2 Polishing Graphics

Visual polish—better paddles, smoother motion, clearer text—is a natural next step once the core game loop works. The simplest improvements stay within character mode:

- choose characters that look more paddle-like or ball-like than generic letters,
- use colour RAM to differentiate the paddle, ball, and walls with distinct colours,
- tidy up the score display so it always appears in the same place and does not scroll.

A more ambitious upgrade is to move the paddle and ball to hardware sprites. In that design, the text screen still shows borders and scores, but the moving objects are drawn by the VIC-II based on sprite position registers. The core game logic—updating positions, checking collisions, managing scores and lives—remains almost identical; only the drawing and erasing routines change. Treat this as an opportunity to refactor your code so that all drawing goes through a few well-chosen subroutines that you can later swap from character-based to sprite-based implementations.

### 8.3 Beyond Pong

The techniques you use for Pong transfer directly to many other genres. Breakout-style games replace the paddle-and-wall combination with rows of bricks to clear; simple shooters swap the ball for projectiles and add moving targets; maze games reuse the same grid and collision ideas for navigation rather than bouncing.

When you are comfortable with the Pong code, try one of the following experiments:

- turn the right-hand wall into a second paddle controlled by simple AI that tracks the ball's vertical position,
- change the court shape or introduce obstacles in the middle of the field,
- add a "practice mode" where the ball's speed increases gradually with the score.

Keep copies of your intermediate versions so you can see how small code changes create different kinds of games. Over time, this collection of experiments becomes your personal C64 laboratory, echoing the spirit of the home-computer era that the main book describes.

## A Full Pong Listing and How to Run It

This appendix collects the complete BASIC listing for the single-player Pong game developed earlier in this workbook and explains exactly how to type, save, and run it in VICE. The code brings together the earlier fragments into a single, runnable program so you can see how the constants, helper functions, game loop, and subroutines fit into one listing.

### A.1 Full BASIC Listing

The following listing assumes the standard C64 text screen at decimal address `1024` and uses the keyboard keys `Q` (up) and `A` (down) to move the paddle. Type it exactly as shown, paying close attention to line numbers, spaces, and punctuation:

```
10 REM === C64 WORKBOOK PONG ===
20 SCR = 1024 : WIDTH = 40 : HEIGHT = 25
30 TOPROW = 1 : BOTROW = 23
40 LEFTCOL = 1 : RIGHTCOL = 38
50 PADDLELEN = 4
60 BALLCHAR = 81 : PADDLECHAR = 81
70 PADDLECOL = LEFTCOL + 1
80 PADDLETOP = INT((TOPROW+BOTROW)/2 - PADDLELEN/2)
90 BALLR = INT((TOPROW+BOTROW)/2)
100 BALLC = INT((LEFTCOL+RIGHTCOL)/2)
110 DR = -1 : DC = 1
120 SCORE = 0 : LIVES = 3

130 REM CLEAR SCREEN AND DRAW COURT
140 PRINT CHR$(147)
150 GOSUB 300   : REM DRAW BORDERS
160 GOSUB 400   : REM INITIAL STATUS LINE
170 GOSUB 900   : REM DRAW INITIAL PADDLE
180 GOSUB 1100  : REM DRAW INITIAL BALL
190 GOTO 2000   : REM JUMP TO MAIN LOOP

300 REM === DRAW BORDERS ===
```

```
310 FOR C = LEFTCOL TO RIGHTCOL
320   POKE SCR + TOPROW*WIDTH + C, 45
330   POKE SCR + BOTROW*WIDTH + C, 45
340 NEXT C
350 FOR R = TOPROW TO BOTROW
360   POKE SCR + R*WIDTH + LEFTCOL, 124
370   POKE SCR + R*WIDTH + RIGHTCOL, 124
380 NEXT R
390 RETURN

400 REM === INITIAL STATUS LINE ===
410 PRINT "SCORE:"; SCORE; "  LIVES:"; LIVES
420 RETURN

1000 REM MAP (ROW, COL) TO SCREEN ADDRESS
1010 DEF FNADDR(R,C) = SCR + R*WIDTH + C

2000 REM === MAIN GAME LOOP SETUP ===
2010 GOSUB 900    : REM ENSURE PADDLE DRAWN
2020 GOSUB 1100   : REM ENSURE BALL DRAWN
2030 GOTO 2100

2100 REM === MAIN GAME LOOP ===
2110 GOSUB 1200   : REM HANDLE INPUT AND REDRAW PADDLE
2120 GOSUB 1300   : REM UPDATE BALL AND REDRAW
2130 FOR T = 1 TO 80 : NEXT T   : REM SIMPLE DELAY
2140 GOTO 2110

900 REM === DRAW PADDLE BASED ON PADDLETOP ===
910 FOR R = TOPROW+1 TO BOTROW-1
920   POKE FNADDR(R, PADDLECOL), 32
930 NEXT R
940 FOR I = 0 TO PADDLELEN-1
950   POKE FNADDR(PADDLETOP+I, PADDLECOL), PADDLECHAR
960 NEXT I
970 RETURN

1100 REM === DRAW BALL AT (BALLR, BALLC) ===
1110 POKE FNADDR(BALLR, BALLC), BALLCHAR
1120 RETURN

1150 REM === ERASE BALL AT (BALLR, BALLC) ===
1160 POKE FNADDR(BALLR, BALLC), 32
1170 RETURN

1200 REM === HANDLE KEYBOARD INPUT FOR PADDLE ===
1210 GET K$ : IF K$ = "" THEN 1280
1220 IF K$ = "Q" AND PADDLETOP > TOPROW+1 THEN PADDLETOP = PADDLETOP-1
1230 IF K$ = "A" AND PADDLETOP+PADDLELEN < BOTROW THEN PADDLETOP = PADDLETOP+1
1240 GOSUB 900
1280 RETURN
```

```
1300 REM === UPDATE BALL POSITION AND HANDLE BOUNCES ===
1310 GOSUB 1150
1320 BALLR = BALLR + DR
1330 BALLC = BALLC + DC

1340 REM BOUNCE OFF TOP AND BOTTOM
1350 IF BALLR <= TOPROW+1 THEN DR = 1 : BALLR = TOPROW+2
1360 IF BALLR >= BOTROW-1 THEN DR = -1 : BALLR = BOTROW-2

1370 REM BOUNCE OFF RIGHT WALL
1380 IF BALLC >= RIGHTCOL-1 THEN DC = -1 : BALLC = RIGHTCOL-2

1390 REM CHECK FOR PADDLE HIT OR MISS ON LEFT SIDE
1400 IF BALLC <= PADDLECOL THEN GOSUB 1500

1410 GOSUB 1100
1420 RETURN

1500 REM === HANDLE COLLISION WITH LEFT SIDE ===
1510 IF BALLR >= PADDLETOP AND BALLR < PADDLETOP+PADDLELEN THEN GOSUB 1700 : RETURN

1520 REM OTHERWISE, THE PLAYER MISSES
1530 LIVES = LIVES - 1
1540 GOSUB 1800              : REM UPDATE STATUS LINE
1550 IF LIVES = 0 THEN GOSUB 1900 : END
1560 BALLR = INT((TOPROW+BOTROW)/2)
1570 BALLC = INT((LEFTCOL+RIGHTCOL)/2)
1580 DR = -1 : DC = 1
1590 RETURN

1700 REM === SUCCESSFUL PADDLE HIT ===
1710 DC = 1                     : REM SEND BALL BACK TO THE RIGHT
1720 SCORE = SCORE + 1
1730 GOSUB 4000                 : REM OPTIONAL HIT SOUND
1740 GOSUB 1800                 : REM UPDATE STATUS LINE
1750 RETURN

1800 REM === PRINT SCORE AND LIVES ===
1810 PRINT "SCORE:"; SCORE; "  LIVES:"; LIVES
1820 RETURN

1900 REM === GAME OVER MESSAGE ===
1910 PRINT : PRINT "GAME OVER"
1920 PRINT "FINAL SCORE:"; SCORE
1930 RETURN

4000 REM === SIMPLE HIT SOUND ON SID VOICE 1 ===
4010 POKE 54276, 17      : REM SET ATTACK/DECAY
4020 POKE 54277, 240     : REM SET SUSTAIN/RELEASE
4030 POKE 54272, 34      : REM FREQUENCY LOW BYTE
```

```
4040 POKE 54273, 8       : REM FREQUENCY HIGH BYTE
4050 POKE 54278, 33      : REM ENABLE PULSE WAVEFORM AND GATE
4060 FOR T = 1 TO 100 : NEXT T
4070 POKE 54278, 32      : REM TURN GATE BIT OFF
4080 RETURN
```

## A.2 Running the Game in VICE

To run the game in the VICE emulator, follow these steps:

1. Start the `x64sc` C64 emulator and wait for the `READY.` prompt.
2. Type `NEW` and press `RETURN` to clear any old program.
3. Type in the listing above line by line. After each line, press `RETURN`. You can use `LIST` at any time to review what you have entered and correct mistakes by retyping a line with the same line number.
4. When you have finished typing, save your work to the current disk image so you do not lose it:

   ```
   SAVE "PONG",8
   ```

   (Adjust the device number if you use a different drive in VICE.)
5. To start the game, type:

   ```
   RUN
   ```

   The program performs its initial setup (clearing the screen, drawing borders, showing the initial score and lives) and then enters the main loop.
6. Move the paddle with `Q` (up) and `A` (down). Each successful hit increments the score; missing the ball reduces your lives. When lives reach zero, a game-over message appears with your final score.

If you later reload the program from disk with:

```
LOAD "PONG",8
RUN
```

you should see the same behaviour. From there, you can start experimenting with the ideas in the "Next Experiments and Extensions" section by modifying the listing and observing how the C64 responds.

## A.3 Transferring the Program from macOS to VICE

In many cases you will prefer to edit the BASIC listing in a text editor on your Mac and then move it into VICE, rather than typing everything at the C64 prompt. There are several practical ways to do this; the following two are simple and rely only on tools most users already have.

**Option 1: Paste into VICE.**

- Open the listing in a plain-text editor and ensure that each BASIC line starts with a line number, exactly as in the appendix.
- Copy a small block of lines (for example 20–30 at a time) to the macOS clipboard.
- Click into the VICE window so that the C64 prompt is active, then paste. VICE will feed the characters to the emulated machine as if you had typed them.
- After each block, use `LIST` to verify that the lines were received correctly before pasting the next block.

This method is quick and works well for modest-sized programs, but it depends on timing and can produce garbled lines if the emulator cannot keep up. Keeping pasted chunks small reduces the risk.

**Option 2: Create a `.prg` file with a BASIC tool.**

- Use a BASIC-aware tool on your Mac to convert the text listing into a C64 `.prg` file. Several open-source utilities can read numbered BASIC lines and produce tokenised programs suitable for loading in VICE.
- Place the resulting `PONG.PRG` into a disk image or a directory that VICE mounts as a drive.
- In VICE, attach that image or directory to drive 8 and load the program with:

```
LOAD "PONG",8
RUN
```

This approach takes a little more setup but makes it easy to keep the authoritative version of your code under version control on the host machine while still running it natively on the virtual C64.

Whichever path you choose, the important idea is that the listing in this appendix can live both as a typed-in program inside the emulator and as a text file on your Mac. Moving comfortably between those worlds mirrors how many retro-computing practitioners work today: editing and organising code on a modern system, then deploying it to emulated or physical 8-bit hardware for execution and debugging.

## References


[1] Y. J. Hilpisch, *A Short History of Computer Science: From Commodore C64 to Cloud AI*, 2025.

[2] Y. J. Hilpisch, *VICE C64 Emulator for macOS: Installation and User Manual*, 2025.