USER MANUAL

C64 IDE

The Complete Guide to C64 IDE

Version 1.2

Gopher Broke Software — 2026

C64 IDE User Manual

Version 1.2

Gopher Broke Software

Table of Contents

Chapters

Appendices

C64 IDE User Manual

Version 1.2

Gopher Broke Software

Chapter 1: Introduction

Welcome to C64 IDE

The Commodore 64 was released in 1982 and went on to become the best-selling personal computer model of all time. Decades later, a passionate global community of developers, artists, and musicians continues to create new software, games, and demos for it — and the tools available to them have never been better.

C64 IDE is a macOS application designed to bring that creative work into a modern development environment without losing touch with what makes C64 development unique. Whether you're writing BASIC programs, hand-crafting 6502 assembly, designing sprites, composing SID music, or building a full game from scratch, C64 IDE gives you the tools to do it all in one place.

This manual will guide you through every feature of the application, from writing your first BASIC program to debugging assembly code live in the VICE emulator.

What's Included

C64 IDE is a complete development environment for the Commodore 64. Here's a quick overview of what it contains:

- **Code Editor** — A full-featured text editor with syntax highlighting for both Commodore BASIC and 6502 assembly, with an integrated reference panel so you never have to leave the app to look something up.
- **Build System** — Integrated ca65 assembler support with a build console that shows errors, warnings, and symbol information. Build and launch directly into VICE with a single keystroke.
- **VICE Debugger** — A source-level debugger that connects to the VICE Commodore emulator, letting you set breakpoints, step through code, inspect registers and memory, and see exactly where your program is executing — highlighted right in the editor.
- **Sprite Editor** — A pixel-art editor for designing C64 sprites, with support for single-color and multi-color modes, animation frames, onion skinning, and direct export to BASIC DATA statements or assembly.
- **Character Set Editor** — Design custom character sets for your programs, with a live character map preview, flip and shift tools, and export to binary or assembly.
- **Hi-Res Graphics Editor** — A 320×200 bitmap drawing canvas with a full set of drawing tools, zoom support, and C64-accurate color constraints.
- **Game Map Editor** — A tile-based map editor that uses your custom character sets to build game worlds, with multiple layers and flood fill support. The Game Map Editor is live-linked to the Character Set Editor — any changes you make to your character set are reflected in the map instantly, in real time.
- **SID Editor** — A tracker-style music editor for the C64's SID sound chip, with ADSR envelope control, three-voice sequencing, and export to assembly or BASIC.
- **Image Converter** — Convert modern images to C64 hi-res or multicolor bitmap format, with dithering options and direct export to D64 disk images.

- **6502 Disassembler** — Load any C64 PRG file and disassemble it, with automatic annotation of KERNAL routines, VIC-II, SID, and CIA registers. Export as ca65-compatible assembly or open directly in the editor.
- **D64 Disk Image Browser** — Open, create, and manage Commodore 1541 disk images. Add, extract, and delete files, or open programs directly in the IDE.
- **PETSCII Character Map** — A full interactive reference for PETSCII character codes, with decimal, hex, binary, and screen code values at a glance.
- **C64 Character ROM Viewer** — Browse both character sets built into the C64 ROM, with full pixel data and code information for every character.
- **Number Converter** — A programmer's calculator with decimal, hex, binary, and octal conversion, clickable bit toggling, bitwise operations, and quick access to common C64 memory addresses.
- **BASIC Keyword/Plugin Editor** — Define custom BASIC extension keywords for enhanced BASIC interpreters, with syntax, descriptions, and JSON export for use as IDE plugins.

System Requirements

- **macOS:** 11.0 (Big Sur) or later
- **Processor:** Intel or Apple Silicon (Universal Binary)
- **VICE Emulator:** Required for running and debugging programs. Available free from vice-emu.sourceforge.io, or via Homebrew (`brew install vice`).
- **cc65/ca65:** Required for assembling 6502 assembly programs. Available free from cc65.github.io, or via Homebrew (`brew install cc65`).

> **Note:** VICE and cc65 are third-party tools and are not included with C64 IDE. Homebrew is the easiest installation method for most users. See Chapter 2 for full installation and setup instructions.

A Note on the Commodore 64

If you're new to C64 development, a few things are worth knowing upfront.

The C64 uses an 8-bit MOS 6510 processor (a variant of the 6502) running at approximately 1 MHz. It has 64KB of RAM, a powerful-for-its-time graphics chip called the VIC-II, and a legendary three-voice sound chip called the SID. Programs are written in either Commodore BASIC V2 (built into ROM) or 6502 assembly language, and they're typically distributed on floppy disk images in the `.d64` format.

Memory addresses are central to C64 programming. Almost everything — setting a color, enabling a sprite, playing a sound — involves reading or writing specific memory locations. C64 IDE's reference tools are designed to keep that information close at hand so you can spend more time coding and less time digging through documentation.

Welcome aboard. Let's build something.

Chapter 2: Getting Started

This chapter walks you through installing C64 IDE and its companion tools, configuring everything to work together, and running your first program on the VICE emulator. By the end you'll have a working development environment and a taste of what C64 IDE can do.

Installing C64 IDE

1. Download the latest version of C64 IDE from gopherbrokesoftware.com.
2. Unzip the downloaded file.
3. Drag **C64 IDE.app** to your Applications folder, or anywhere else you'd like to keep it.
4. Double-click to launch. If macOS warns you that the app was downloaded from the internet, click **Open** to proceed.

That's it. C64 IDE requires no installer.

C64 IDE checks for updates once a day and will notify you when a new version is available. If you see an update prompt shortly after launching, that's normal — just follow the prompts to stay current.

Installing VICE and cc65

C64 IDE needs two companion tools to build and run programs:

- **VICE** — a Commodore 64 emulator. C64 IDE uses its `x64sc` binary to run your programs and its built-in monitor for debugging.
- **cc65** — a cross-development package for 6502-based systems. C64 IDE uses its `ca65` assembler and `ld65` linker to build assembly programs.

Neither tool is required just to edit files, but you'll need both to build and run anything.

INSTALLING VIA HOMEBREW (RECOMMENDED)

Homebrew is a free package manager for macOS and the easiest way to install both tools. If you don't have Homebrew installed, visit brew.sh and follow the instructions there first.

Once Homebrew is installed, open Terminal and run:

```
brew install vice
brew install cc65
```

Homebrew will install both tools and place them in `/opt/homebrew/bin/` on Apple Silicon Macs, or `/usr/local/bin/` on Intel Macs. C64 IDE will find them automatically — no further configuration needed.

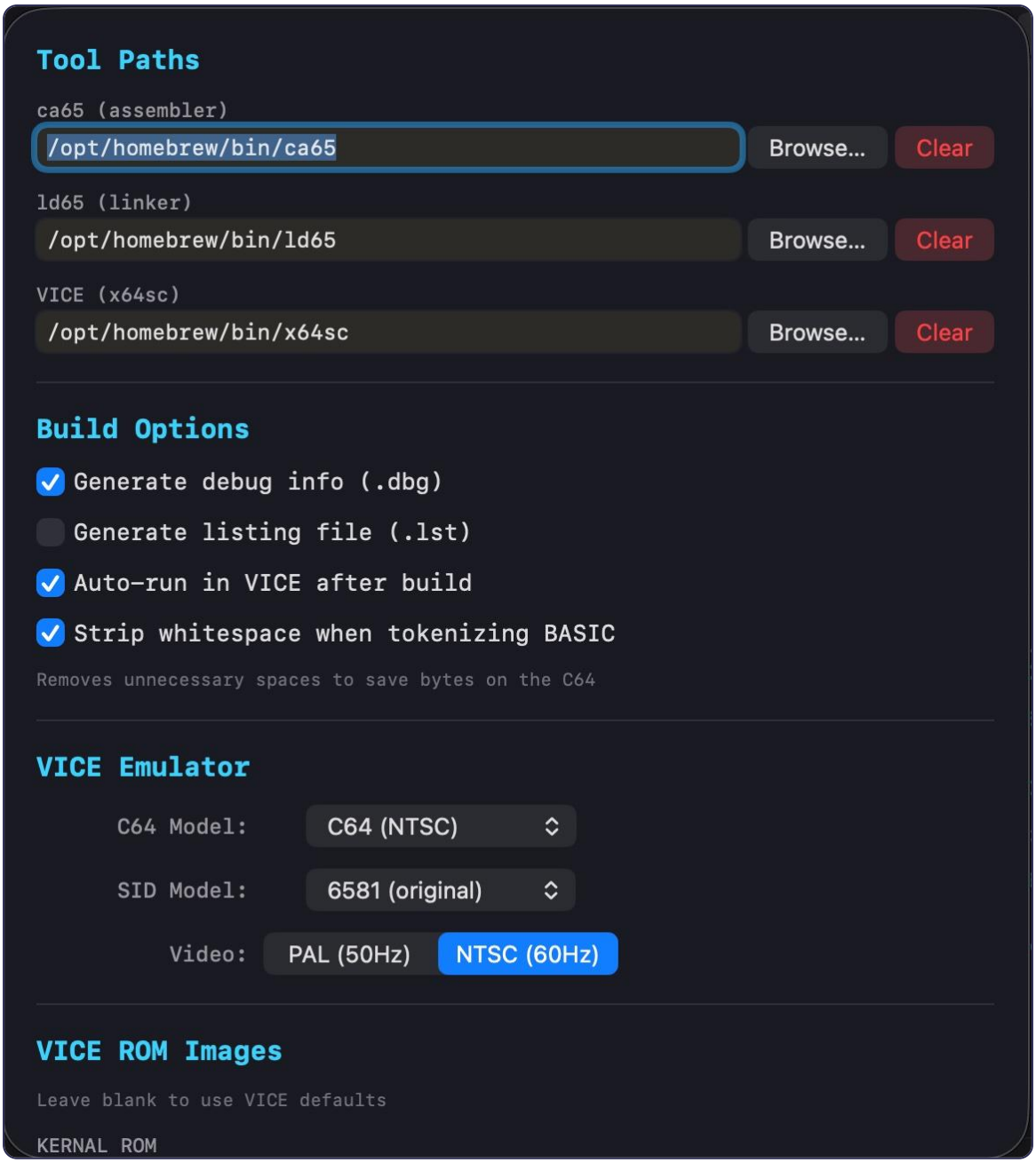
INSTALLING MANUALLY

If you prefer to install VICE or cc65 manually:

- **VICE:** Download the macOS build from vice-emu.sourceforge.io. Note the location of the `x64sc` binary inside the application bundle or bin folder.

- **cc65:** Download the latest release from cc65.github.io. Note the locations of the `ca65` and `ld65` binaries.

You'll then need to enter these paths manually in C64 IDE's Settings window, described in the next section.



Configuring C64 IDE

Open Settings by pressing ⌘**,** or choosing **C64 IDE → Settings** from the menu bar.

TOOL PATHS

The top section of Settings shows the paths to ca65, ld65, and x64sc. If you installed via Homebrew, these will already be filled in correctly. If you installed manually, click **Browse…** next to each field to locate the binaries, or type the paths directly.

The **Auto-Detect Paths** button at the bottom of the Settings window will search common locations and fill in any paths it can find automatically.

BUILD OPTIONS

OPTION	DEFAULT	DESCRIPTION
Generate debug info (.dbg)	On	Generates a debug symbol file used by the VICE Debugger for source-level debugging. Recommended to leave on.
Generate listing file (.lst)	Off	Generates a human-readable assembly listing file alongside the build output.
Auto-run in VICE after build	On	Automatically launches your program in VICE after a successful build.
Strip whitespace when tokenizing BASIC	On	Removes unnecessary spaces from BASIC programs when tokenizing, saving bytes on the C64.

VICE EMULATOR

SETTING	DESCRIPTION
C64 Model	The C64 hardware variant to emulate.
SID Model	Choose between the 6581 (the original SID chip, with its characteristic warm sound) and the 8580 (the revised chip found in later C64 models).
Video	PAL (50Hz, common in Europe) or NTSC (60Hz, common in North America). This affects timing-sensitive programs. Choose whichever matches your target audience or your own hardware.

VICE ROM IMAGES

By default, VICE uses its own built-in ROM images. If you have original ROM dumps from a real C64, you can point C64 IDE to them here using the **Browse...** buttons. Leave these fields blank to use VICE's defaults, which work fine for most purposes.

Click **Save** when you're done.

Your First BASIC Program

With everything configured, let's write a simple BASIC program and run it in VICE.

STEP 1: CREATE A NEW FILE

Launch C64 IDE and choose **File → New → New BASIC File**. A new untitled tab will open in the Code Editor.

STEP 2: WRITE THE PROGRAM

Type the following into the editor:

```
10 REM *** HELLO, C64! ***
20 PRINT CHR$(147)
30 PRINT "HELLO FROM C64 IDE!"
40 PRINT
50 PRINT "THE YEAR IS 1984."
60 END
```

Line 20 uses `CHR$(147)` to clear the screen before printing — a common C64 idiom you'll see often.

STEP 3: SAVE THE FILE

Press ⌘**S** to save. Give the file a name ending in `.bas`, for example `hello.bas`. Choose a folder you'll remember.

STEP 4: BUILD AND RUN

Press ⌘**R** (or click the **Run** button in the toolbar). If you haven't saved the file yet, C64 IDE will prompt you to do so before building. Once saved, it will:

1. Tokenize your BASIC program into C64 binary format.
2. Launch VICE with the program loaded and ready to run.

VICE will open and your program will execute automatically. You should see **HELLO FROM C64 IDE!** printed on the familiar blue C64 screen.

STEP 5: CHECK THE BUILD CONSOLE

At the bottom of the Code Editor window, the **Build** tab shows the output from the build process. After a successful BASIC build you'll see confirmation that the program was tokenized and the VICE launch command. If there are any errors, they'll appear here with details about what went wrong.

Your First Assembly Program

If you're familiar with 6502 assembly, here's a minimal example to confirm your assembler is working correctly.

STEP 1: CREATE A NEW FILE

Choose **File → New → New Assembly File** and save it as `hello.s`.

STEP 2: WRITE THE PROGRAM

```
; hello.s - Minimal C64 assembly example

CHROUT = $FFD2      ; KERNAL routine: output character to screen

.export __LOADADDR__: absolute = 1

.segment "LOADADDR"
    .word $0801             ; Load address

.segment "STARTUP"
    ; BASIC stub: 10 SYS 2061
    .word @end
    .word 10
    .byte $9E               ; BASIC token for SYS
    .byte "2061", 0         ; Address of our code as ASCII string
@end:
    .word 0                 ; End of BASIC program

.segment "CODE"
_start:
```

```
    ldx #0
print_loop:
    lda message,x
    beq done
    jsr CHROUT
    inx
    bne print_loop
done:
    rts

message:
    .byte "hello from c64 ide!", 13, 0
```

STEP 3: BUILD AND RUN

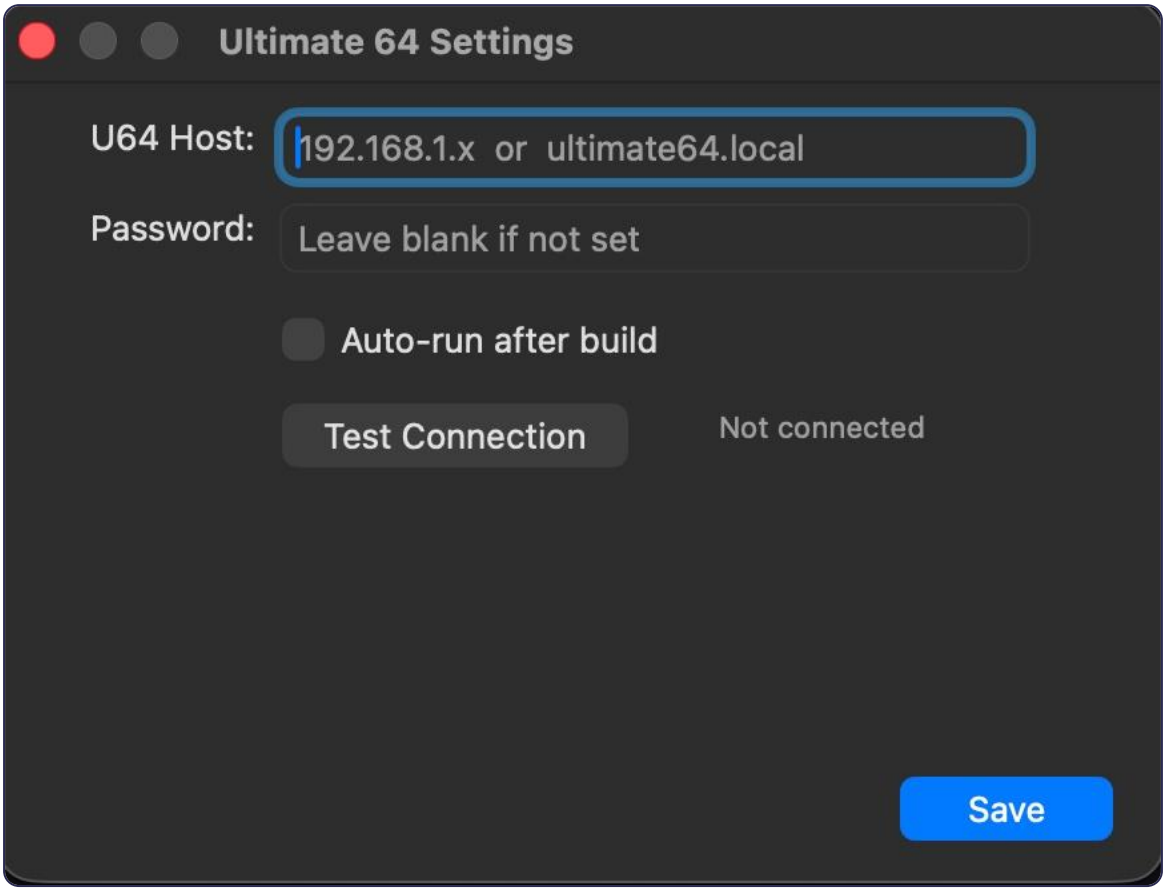
Press ⌘**R**. C64 IDE will assemble and link the program using ca65 and ld65, then launch it in VICE. The Build console will show the number of line mappings and symbols generated, confirming that debug information is available for the VICE Debugger.

The Toolbar

The Code Editor toolbar contains five buttons:

BUTTON	SHORTCUT	DESCRIPTION
Run	⌘R	Builds your program and launches it in VICE. Prompts to save first if needed.
Stop	⌘.	Terminates VICE immediately.
Run on U64		Builds your program and sends it to a real Ultimate 64 over the network.
Reference	⌘⌥R	Shows or hides the Reference panel on the right side of the editor.

BUTTON	SHORTCUT	DESCRIPTION
Console	⌘⇧Y	Shows or hides the Build console at the bottom of the editor.



Running on a Real Ultimate 64

If you're lucky enough to own an Ultimate 64 — a modern FPGA-based C64 motherboard replacement — or one of the brand new C64s now being produced (which are based on the same Ultimate 64 FPGA core), C64 IDE can build your program and send it directly to the real hardware over your local network, no floppy disk or SD card required.

SETTING UP U64 SUPPORT

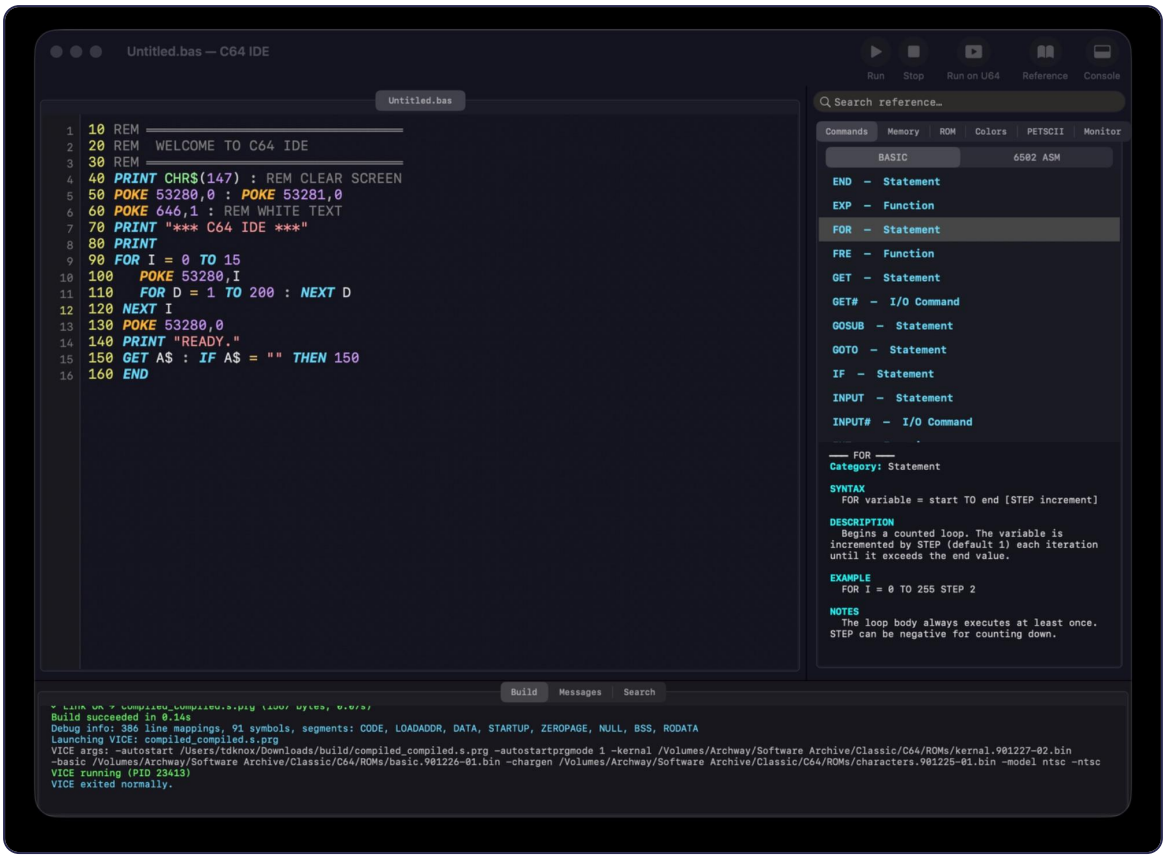
1. Make sure your Ultimate 64 is powered on and connected to your local network.
2. In C64 IDE, choose **Tools → U64 Settings** from the menu bar.

3. Enter your U64's IP address (e.g. `192.168.1.x`) or hostname (`ultimate64.local`) in the **U64 Host** field.
4. If your Ultimate 64 has a password set, enter it in the **Password** field. Otherwise leave it blank.
5. Click **Test Connection** to confirm C64 IDE can reach the hardware.
6. Optionally enable **Auto-run after build** to have the program start automatically on the U64 after each successful build.
7. Click **Save**.

Once configured, click **Run on U64** in the toolbar to build and deploy to your real hardware. There's something uniquely satisfying about hitting a button on your Mac and watching your code run on actual C64 silicon.

What's Next?

Now that everything is up and running, the rest of this manual covers each feature of C64 IDE in detail. If you want to dive straight into coding, head to **Chapter 3: The Code Editor**. If you'd like to understand the build system before going further, **Chapter 5: Build System** has you covered.



Chapter 3: The Code Editor

The Code Editor is the heart of C64 IDE. It's where you write, edit, and build your programs — whether you're working in Commodore BASIC or 6502 assembly. It combines a full-featured text editor with an integrated reference panel, a build console, and direct access to the VICE debugger, so everything you need is within reach without leaving the app.

Opening and Managing Files

CREATING NEW FILES

Use the **File → New** submenu to create a new file:

MENU ITEM	DESCRIPTION
File → New → New BASIC File	Creates a new Commodore BASIC source file (`.bas`)

MENU ITEM	DESCRIPTION
File → New → New Assembly File	Creates a new 6502 assembly source file (`.s`), pre-populated with a working BASIC stub and CODE segment ready to build
File → New → New Map	Opens a new Game Map Editor window

TABS

C64 IDE supports multiple open files simultaneously, each in its own tab along the top of the editor window. Click any tab to switch to that file. To close a tab, click the close button on the tab itself.

> **Note:** Tab reordering is not currently supported.

SAVING FILES

Press ⌘**S** to save the current file. If the file has not been saved before, you'll be prompted to choose a name and location. C64 IDE will also prompt you to save automatically if you attempt to build an unsaved file.

Syntax Highlighting

The editor automatically applies syntax highlighting based on the file type:

- **BASIC files** (`.bas`) — BASIC keywords, line numbers, strings, and comments are highlighted in distinct colors, making the structure of your program easy to read at a glance. Keywords are recognized even when written without surrounding spaces, as is common in compact C64 BASIC code — for example `FORI=0TO15` and `POKE53281,0` are highlighted correctly.
- **Assembly files** (`.s`) — 6502 mnemonics, directives, labels, comments, and operands are each highlighted distinctly.

The editor does not mix highlighting modes within a single file — each file is treated as either BASIC or assembly based on its extension.

The Reference Panel

The Reference panel lives on the right side of the Code Editor window and puts the entire C64 programming reference at your fingertips. Toggle it open or closed with the **Reference** button in the toolbar, or by pressing ⌘⌥**R**.

The panel has two modes, selectable at the top:

- **BASIC** — reference entries for all Commodore BASIC V2 keywords
- **6502 ASM** — reference entries for all 6502 assembly mnemonics

REFERENCE TABS

Within each mode, the panel is organized into tabs:

TAB	CONTENTS
Commands	Alphabetical list of keywords or mnemonics with brief descriptions
Memory	Common C64 memory locations and their functions
ROM	KERNAL routine addresses and descriptions
Colors	C64 color values and their names
PETSCII	PETSCII character codes
Monitor	VICE monitor command reference

Every tab is fully searchable using the search field at the top of the Reference panel. Type any part of a keyword, address, or description and the list will filter instantly to matching entries.

CLICKING KEYWORDS

Click any keyword or mnemonic in the editor and the Reference panel will automatically switch to the correct mode and scroll directly to that keyword's entry. The entry shows the full reference information: category, syntax, description, a usage example, and any relevant notes.

HOVER TOOLTIPS

Hover the mouse over any keyword or mnemonic in the editor and a tooltip will appear showing the complete reference entry inline — the same information as the Reference panel, without having to look away from your code. This is particularly handy when you just need a quick syntax reminder.

Breakpoints

Click any line number in the gutter to set a breakpoint on that line. A red dot will appear next to the line number to indicate an active breakpoint. Click the dot again to clear it.

Breakpoints are used with the **Build and Debug** command (⌘⇧R). When you build and debug, C64 IDE launches VICE, opens the VICE Debugger window, connects automatically, and pauses execution at the start of your program — ready for you to step through your code or click **Continue** to run to the first breakpoint. See **Chapter 6: VICE Debugger** for full details.

Building Your Program

C64 IDE offers several build commands depending on what you want to do with your program after it's built. All build commands will prompt you to save the file first if there are unsaved changes.

COMMAND	SHORTCUT	DESCRIPTION
Build and Run	⌘R	Builds the program and launches it in VICE.
Build and Debug	⌘⇧R	Builds the program, launches VICE, opens the VICE Debugger, connects automatically, and pauses at the start of the program.
Build Only	⌘B	Builds the program without launching VICE. Useful for checking for errors without running.
	⌘⌥D	

COMMAND	SHORTCUT	DESCRIPTION
Build and Save to D64		Builds the program and saves it to a D64 disk image. A dialog will ask whether to create a new image or add to an existing one.
Compile BASIC to ASM	⌘⇧G	Compiles the current BASIC program to ca65-compatible assembly and opens the result in a new editor tab. See below for details.
Run on U64	⌘⌃R	Builds the program and sends it to a connected Ultimate 64 or new-production C64, launching it automatically.
Load on U64	⌘⌃L	Builds the program and transfers it to a connected Ultimate 64 or new-production C64, but does not run it — leaving it ready for you to start manually.
Stop	⌘.	Terminates VICE immediately.

WHAT HAPPENS DURING A BASIC BUILD

When you build a BASIC file, C64 IDE tokenizes your source code into the binary format the C64 expects — the same format it would produce if you typed the program directly into the machine. If **Strip whitespace when tokenizing BASIC** is enabled in Settings, unnecessary spaces are removed to save bytes on the C64.

WHAT HAPPENS DURING AN ASSEMBLY BUILD

When you build an assembly file, C64 IDE runs `ca65` to assemble your source code, then `ld65` to link it into a C64 PRG file. If **Generate debug info (.dbg)** is enabled in Settings, a `.dbg` file is also produced alongside the PRG. This file is used by the VICE Debugger for source-level debugging — mapping machine code addresses back to lines in your source file.

Compiling BASIC to Assembly

Compile BASIC to ASM (⌘⇧G) is one of C64 IDE's more powerful features. It takes your BASIC program and compiles it into ca65-compatible 6502 assembly language, opening the result as a new editor tab.

For example, if your source file is `invaders.bas`, the compiled output will open as `invaders.asm`. The resulting assembly file is immediately buildable with ⌘R — no manual editing required to get it running.

This is useful for several reasons:

- **Learning** — see exactly what the C64 is doing under the hood when it executes your BASIC program.
- **Optimization** — use the compiled assembly as a starting point for hand-optimization of performance-critical code.
- **Migration** — convert a BASIC prototype into assembly for a production version of your program.

The Build Console

The build console sits at the bottom of the Code Editor window and has three tabs:

BUILD TAB

Shows the raw output from the build tools — ca65, ld65, and the VICE launch command. A successful assembly build will show something like:

```
Building: hello.s
═══════════════════════════════════════
Assembling with ca65...
✓ Assembly OK (0.10s)
Linking with ld65...
✓ Link OK (0.05s)
```

```
Build info: 386 line mappings, 91 symbols
Launching VICE: hello.prg
```

Errors and warnings from ca65 and ld65 appear here with file names and line numbers, making it easy to track down problems.

MESSAGES TAB

Shows status messages from C64 IDE itself, with timestamps:

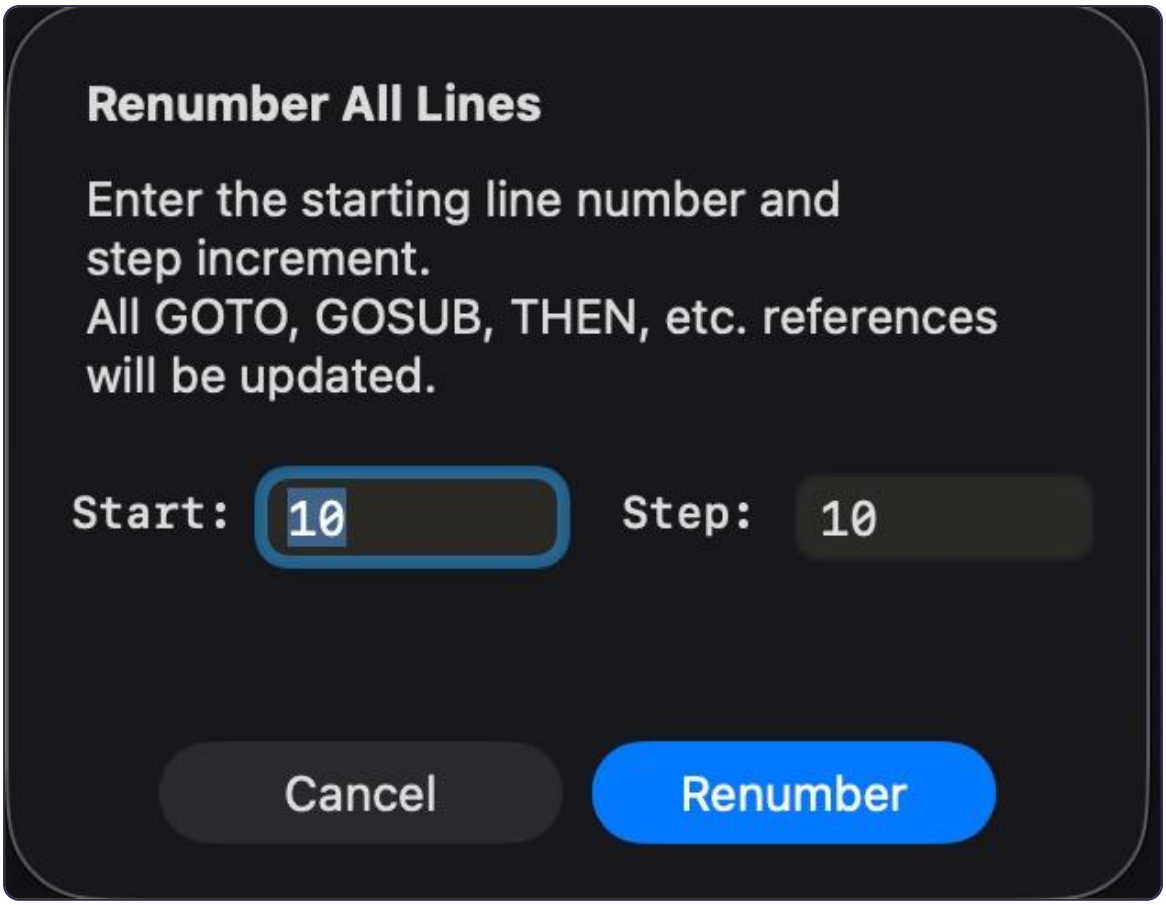
```
[6:21:48 PM] Build failed: 1 error, 2 warnings
[6:32:52 PM] Build succeeded in 0.01s
```

SEARCH TAB

The Search tab allows you to search and replace across all open files. Enter a search term in the search field and matching text will be highlighted across all open tabs. Use the Replace field to substitute matches individually or all at once.

Undo and Redo

The editor supports up to 10 levels of undo and redo. Press ⌘**Z** to undo and ⌘⇧**Z** to redo. Undo history is per-file and is maintained independently for each open tab.



The Edit Menu

Beyond the standard cut, copy, paste, and select all commands, the Edit menu includes:

Edit → Renumber BASIC Lines — Opens a dialog where you specify a starting line number and step increment (both defaulting to 10). C64 IDE will renumber all lines in the program accordingly and automatically update all `GOTO`, `GOSUB`, `THEN`, and other line number references throughout the code. This is invaluable when you've run out of room between existing line numbers and need to make space — a very common situation in C64 BASIC development.

Chapter 4: The Build System

C64 IDE's build system handles everything between saving your source code and running your program in VICE — tokenizing BASIC, assembling and linking 6502 assembly, generating debug information, and organizing the output

files. This chapter explains what happens under the hood so you know what to expect and how to get the most out of it.

Overview

C64 IDE supports two distinct build pipelines depending on the type of source file you're working with:

- **BASIC pipeline** — Your `.bas` source file is tokenized internally by C64 IDE into the binary format the C64 expects, and the result is saved as a `.prg` file ready to run.
- **Assembly pipeline** — Your `.s` source file is assembled by `ca65` and linked by `ld65` into a `.prg` file, with optional debug and listing output generated alongside it.

In both cases the end result is a `.prg` file — a standard Commodore 64 program file with a two-byte load address header, exactly as the C64 expects it.

The Build Folder

When you build a program, C64 IDE automatically creates a `build` folder in the same directory as your source file. All build output goes into this folder:

```
my-project/
    hello.bas          ← your source file
    build/
        hello.prg      ← the built program
        hello.dbg      ← debug symbol information
        hello.mon      ← source-to-binary line mappings
        hello.lst      ← assembly listing (if enabled)
```

Keeping build output separate from your source files makes project management clean and straightforward. If you ever want to force a

completely fresh build, simply delete the `build` folder and build again — your source files are untouched.

> **Note:** C64 IDE does not yet support multi-file projects. Each build processes a single source file. The active tab in the editor is what gets built.

The BASIC Pipeline

When you build a BASIC file, C64 IDE's internal tokenizer converts your source code into the binary token format used by the C64's BASIC interpreter. This is the same format the C64 produces when you type a program directly into the machine.

HOW TOKENIZATION WORKS

Commodore BASIC stores programs in a compact binary format rather than plain text. Each BASIC keyword — `PRINT`, `FOR`, `POKE`, and so on — is replaced by a single token byte. Line numbers are stored as 16-bit integers. The tokenizer handles all of this automatically, including keywords that appear without surrounding spaces, such as `FORI=0TO15` or `POKE53281,0`.

WHITESPACE STRIPPING

If **Strip whitespace when tokenizing BASIC** is enabled in Settings, the tokenizer removes unnecessary spaces from your source before tokenizing. Since spaces between tokens are not required by the C64's BASIC interpreter, this can meaningfully reduce the size of your program — useful on a machine with 64KB of RAM to share between the operating system, your program, and its variables.

Your source file on disk is never modified — whitespace stripping only affects the tokenized output in the build folder.

BASIC BUILD OUTPUT

FILE	DESCRIPTION
`program.prg`	The tokenized BASIC program, ready to load and run on the C64 or in VICE.

The Assembly Pipeline

When you build an assembly file, C64 IDE runs two external tools in sequence: `ca65` to assemble your source code, and `ld65` to link the assembled output into a final PRG file.

STAGE 1: ASSEMBLY (CA65)

`ca65` is the assembler from the cc65 cross-development package. It reads your `.s` source file and produces an object file containing the assembled machine code, along with symbol and debug information.

C64 IDE manages the ca65 configuration entirely — you never need to write or edit a linker configuration file. The correct settings for producing a C64 PRG file are handled automatically.

STAGE 2: LINKING (LD65)

`ld65` is the linker from the cc65 package. It takes the object file produced by ca65, resolves addresses and symbols, and produces the final `.prg` file. It also generates the debug and monitor files used by the VICE Debugger.

ASSEMBLY BUILD OUTPUT

FILE	DESCRIPTION
`program.prg`	The assembled and linked program, ready to run on the C64 or in VICE.
`program.dbg`	Debug symbol information — variable names, code segments, and breakpoint data. Used by the VICE Debugger.
`program.mon`	

FILE	DESCRIPTION
	Source-to-binary line mappings — ties each line of your source code to its address in the PRG file, keeping the VICE Debugger and the IDE in sync when stepping through code.
`program.lst`	Assembly listing file (generated only if **Generate listing file** is enabled in Settings). A human-readable view of the assembled code showing each source line alongside its address and generated bytes — useful for verifying exactly what code is being produced.

Both the `.dbg` and `.mon` files must be present for source-level debugging to work correctly. If either file is missing or out of date, the VICE Debugger may not be able to map execution back to your source lines. Rebuilding your program will always regenerate both files.

Error Navigation

If your build produces errors or warnings, they appear in the **Build** tab of the build console with the source file name and line number. Click any error or warning in the console to jump directly to that line in the editor — no need to manually hunt down the offending code.

Warnings do not prevent a successful build but are worth investigating. Errors will stop the build and no output files will be produced.

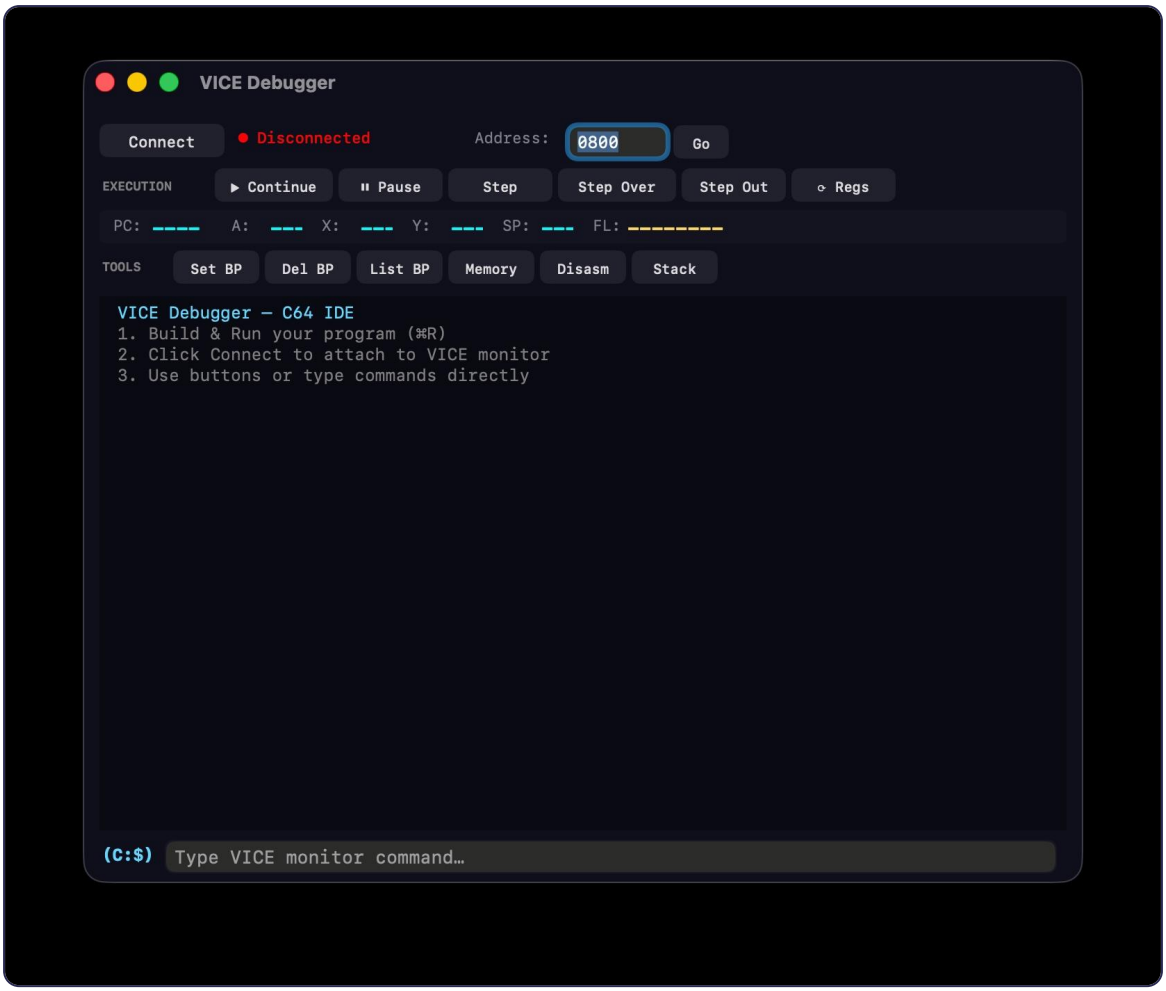
Build Settings

The following settings in **C64 IDE → Settings** affect the build system:

SETTING	EFFECT
Generate debug info (.dbg)	Generates the `.dbg` and `.mon` files required for source-level debugging in the VICE Debugger. Recommended to leave enabled unless you specifically need a smaller build output.

SETTING	EFFECT
Generate listing file (.lst)	Generates a human-readable assembly listing alongside the build output. Disabled by default.
Auto-run in VICE after build	Automatically launches your program in VICE after a successful build when using **Build and Run** (⌘R).
Strip whitespace when tokenizing BASIC	Removes unnecessary spaces from BASIC programs during tokenization to reduce program size.

See **Chapter 2: Getting Started** for the full Settings reference including tool paths and VICE emulator options.



Chapter 5: The VICE Debugger

The VICE Debugger window gives you source-level debugging of your C64 programs running in the VICE emulator. You can step through your code instruction by instruction, inspect and modify registers and memory, set breakpoints, and watch the current execution point highlight in real time — both in the debugger window and back in your source code in the editor.

Starting a Debugging Session

The easiest way to start debugging is to use **Build and Debug** (⌘⇧R) from the Code Editor. This will:

1. Build your program, generating the `.dbg` and `.mon` files required for source-level debugging.

2. Launch VICE with the remote monitor enabled.
3. Open the VICE Debugger window.
4. Connect automatically to VICE.
5. Pause execution at the start of your program, ready for you to step through or inspect.

When a debugging session starts, the Reference panel in the Code Editor automatically switches to the **Monitor** tab, putting the full VICE monitor command reference right where you need it.

CONNECTING MANUALLY

You can also connect to a VICE instance that is already running, provided it was launched with the remote monitor option enabled. Click the **Connect** button in the VICE Debugger window to establish a connection — the debugger always connects to localhost on VICE's default monitor port. If the connection is lost — for example if VICE was restarted — click **Connect** again to reconnect.

The connection status indicator next to the Connect button shows whether the debugger is currently connected (green) or disconnected (red).

The Debugger Window

REGISTERS

The register display shows the current state of the 6510's registers after every command:

REGISTER	DESCRIPTION
PC	Program Counter — the address of the next instruction to be executed
A	Accumulator
X	X index register
Y	Y index register

REGISTER	DESCRIPTION
SP	Stack Pointer
FL	Processor flags (N V - B D I Z C)

The registers update automatically after every step or command. In the rare case that the display gets out of sync, click the **Regs** button to force an immediate refresh.

THE ADDRESS FIELD

The address field displays the start address of your program code by default. To move execution to a specific address, type the address into the field as a plain hexadecimal number (e.g. `0810` — no `$` prefix) and either press Return or click **Go**. This sets the Program Counter to that address, so the next instruction executed will be at that location.

THE COMMAND INPUT

At the bottom of the debugger window is a command input field, pre-filled with the VICE monitor prompt `(C:$)`. You can type any VICE monitor command directly into this field and press Return to execute it. The output appears in the main debugger text area above. This gives you full access to the VICE monitor's capabilities beyond what the toolbar buttons provide.

Refer to the **Monitor** tab in the Reference panel for a full list of available VICE monitor commands and their syntax.

Execution Controls

BUTTON	SHORTCUT	DESCRIPTION
Continue		Resume execution until the next breakpoint or a manual pause.
Pause		Pause execution at the current instruction.
Step		

BUTTON	SHORTCUT	DESCRIPTION
		Execute a single instruction and pause. Steps into subroutine calls (`JSR`).
Step Over		Execute a single instruction and pause. If the instruction is a `JSR`, executes the entire subroutine and pauses on return.
Step Out		Execute until the current subroutine returns (`RTS`), then pause.

After each step, the debugger window scrolls to show the current position, the register display updates, and the editor highlights the corresponding source line — making it easy to follow execution through your code.

Breakpoints

Breakpoints can be set in two ways:

FROM THE EDITOR

Click any line number in the Code Editor gutter to set a breakpoint on that line. A red dot appears to confirm it's set. Click the dot again to clear it. Breakpoints set this way are automatically communicated to VICE during a debugging session.

FROM THE DEBUGGER WINDOW

Use the breakpoint buttons in the **Tools** row to manage breakpoints by address:

BUTTON	DESCRIPTION
Set BP	Set a breakpoint at a specific address in VICE.
Del BP	Delete a breakpoint at a specific address.
List BP	List all currently active breakpoints, including those set from the editor.

Tools

BUTTON	DESCRIPTION
Memory	Dumps the next $30 bytes of memory from the current address as a hex display. Useful for inspecting data structures, screen memory, sprite data, and other memory-mapped values.
Disasm	Disassembles the output currently shown in the debugger window back into readable 6502 mnemonics. Useful when the raw monitor output is showing hex values you want to interpret as code.
Stack	Dumps the current contents of the 6502 hardware stack. Useful for understanding subroutine call depth and diagnosing stack overflow issues.

Source-Level Debugging

When your program is built with **Generate debug info (.dbg)** enabled, C64 IDE uses the `.dbg` and `.mon` files to maintain a live mapping between the machine code running in VICE and the source lines in your editor.

As you step through your program:

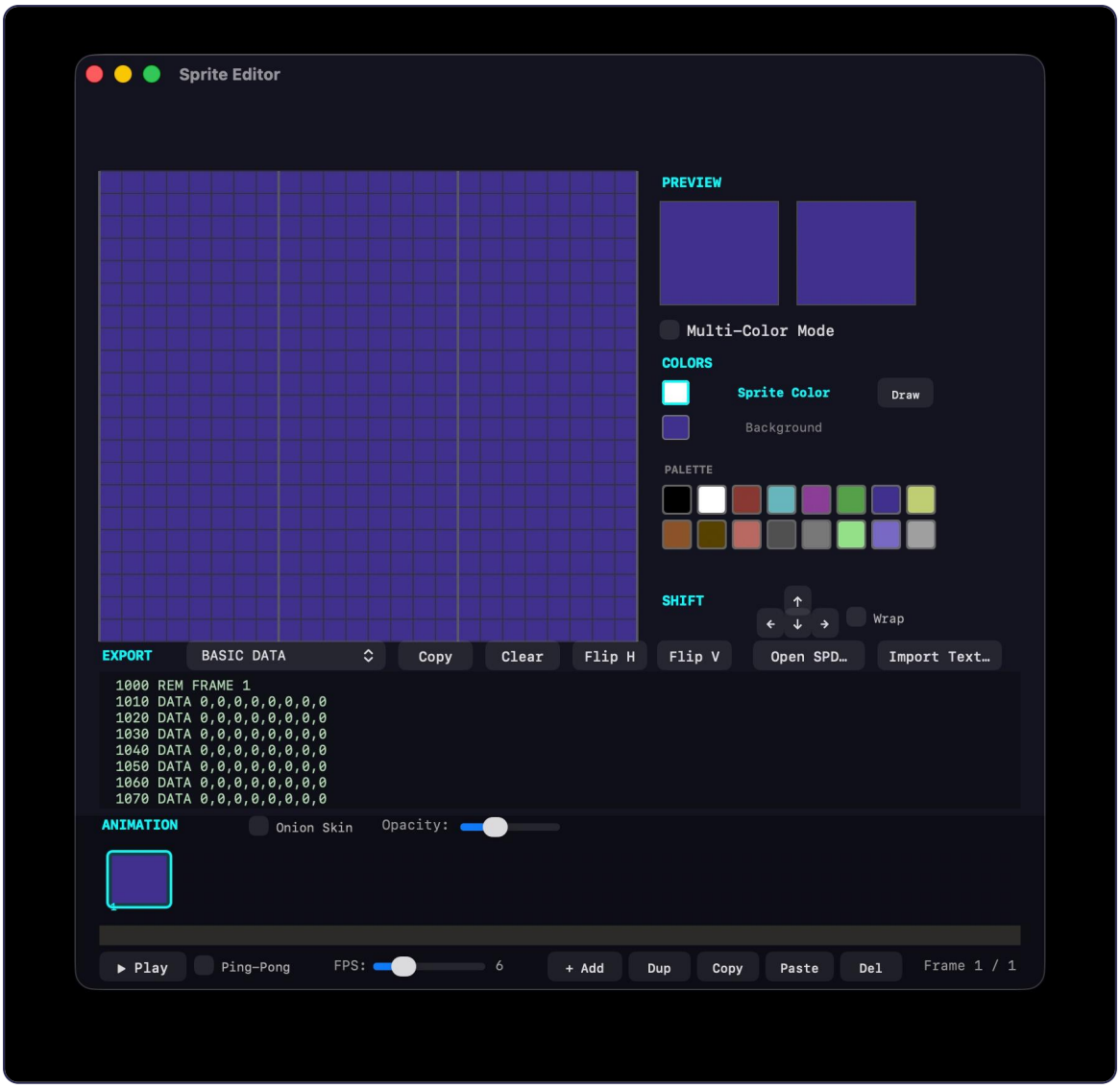
- The debugger window scrolls to follow the current execution point.
- The Code Editor scrolls to the corresponding source line and highlights it.

This two-way synchronization means you're always looking at both the low-level machine state in the debugger and the high-level source context in the editor at the same time — without having to manually track where you are in either.

If source-level highlighting stops working, check that the `.dbg` and `.mon` files in the `build` folder are up to date. If in doubt, do a fresh build with ⌘B before starting a new debugging session.

Tips for Effective Debugging

- **Use Build and Debug (⌘⇧R) rather than Build and Run (⌘R)** when you intend to debug. It ensures the debug files are fresh and the debugger connects automatically.
- **Set breakpoints before building.** Breakpoints set in the editor are included in the debug session automatically — you don't need to set them manually in the VICE monitor.
- **Use Step Over for subroutines you trust.** If you know a KERNAL routine or your own subroutine is working correctly, Step Over lets you skip past it without stepping through every instruction inside.
- **Use Step Out to escape a subroutine.** If you accidentally step into a routine you didn't mean to, Step Out will run to the end of it and return you to the caller.
- **Check the Monitor tab in the Reference panel.** When a debugging session is active the Reference panel switches to the Monitor tab automatically, giving you quick access to VICE monitor command syntax without leaving the app.
- **Type VICE monitor commands directly** in the command input for anything the toolbar buttons don't cover — memory searches, register writes, watchpoints, and more.



Chapter 6: The Sprite Editor

The Sprite Editor lets you design, animate, and export C64 sprites without leaving C64 IDE. It supports both single-color and multi-color sprite modes, multi-frame animation with onion skinning, and flexible import and export options that work directly with your BASIC or assembly code.

Open the Sprite Editor from the **Tools** menu or by pressing ⌘⇧**E**.

The Drawing Canvas

The large grid on the left is your drawing canvas. The C64 sprite format is 24×21 pixels, displayed here at a comfortable zoom level for pixel-accurate editing.

- **Click** a pixel to toggle it on or off.
- **Click and drag** to draw continuously across multiple pixels without having to click each one individually.

There are no additional drawing tools such as lines, rectangles, or ovals — the sprite canvas is intentionally kept simple, reflecting the pixel-by-pixel nature of C64 sprite design.

The following operations are available below the canvas:

BUTTON	DESCRIPTION
Flip H	Flips the entire sprite horizontally.
Flip V	Flips the entire sprite vertically.
Clear	Clears all pixels in the current frame, leaving it blank.

All edits support full undo and redo (up to 10 levels) with ⌘**Z** and ⌘⇧**Z**.

Preview

Two preview boxes in the upper right show your sprite as it will appear on screen:

- **Left box** — 1× size preview, showing the sprite at its true C64 pixel dimensions.
- **Right box** — 2× size preview, showing the sprite scaled up for easier viewing.

Both previews update in real time as you draw.

Color Modes

SINGLE-COLOR MODE

In single-color mode the sprite uses two colors: the **Sprite Color** (the drawn pixels) and the **Background** color (the transparent areas). Click the **Sprite Color** or **Background** label to select the active color attribute, then click a color swatch in the palette below to assign it.

MULTI-COLOR MODE

Enable **Multi-Color Mode** with the checkbox to switch the sprite to the C64's multi-color sprite format. In this mode the sprite uses three colors:

- **Shared Color** — a color shared across all multi-color sprites on screen, set globally.
- **Sprite Color** — the per-sprite color, unique to this sprite.
- **Background** — the transparent background color.

Click the **Shared Color**, **Sprite Color**, or **Background** label to make it the active color attribute, then click a color swatch in the palette to assign it.

> **Note:** Multi-color sprites have half the horizontal resolution of single-color sprites — each pixel in the editor represents a 2×1 block on screen. This is a hardware characteristic of the C64's VIC-II chip.

The Palette

The palette displays all 16 C64 colors. Click any color swatch to assign it to the currently selected color attribute (Sprite Color, Background, or Shared Color in multi-color mode).

Shift

The **Shift** controls move all pixel data in the sprite one pixel in the chosen direction — up, down, left, or right. This is useful for fine-tuning the position of your sprite design within the canvas.

By default, pixels that shift off the edge of the canvas are lost. Enable the **Wrap** checkbox to have pixels that fall off one edge reappear on the opposite edge instead.

Export

The export area at the bottom of the window shows the sprite data in your chosen format, ready to paste directly into your code. Use the format dropdown to select the output format:

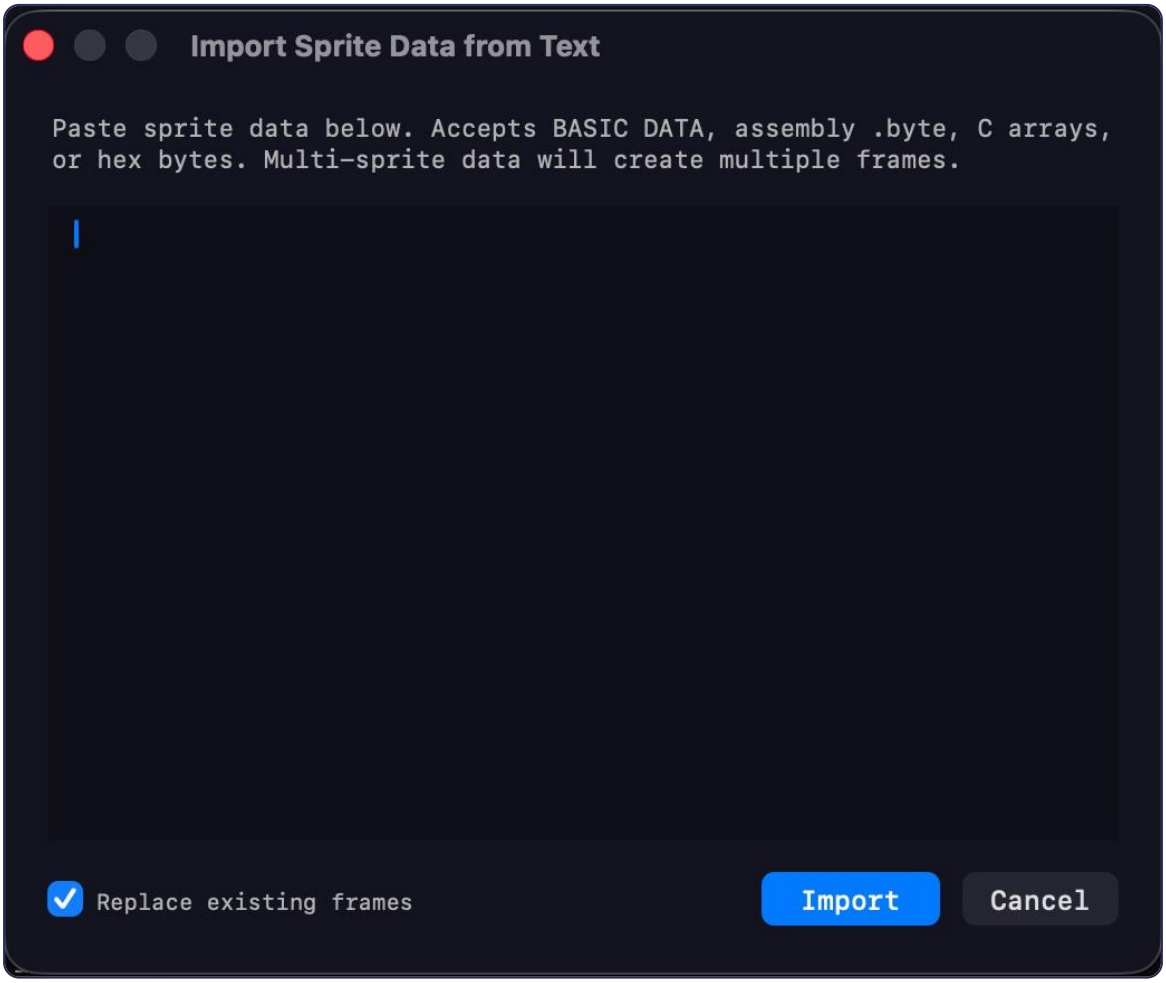
FORMAT	DESCRIPTION
BASIC DATA	Exports as numbered BASIC `DATA` statements, ready to paste into a BASIC program and read with `READ`.
Assembly .byte	Exports as ca65 `.byte` directives, ready to paste into an assembly source file.
C Array	Exports as a C array initializer.
Hex String	Exports as raw hexadecimal byte values.

Click **Copy** to copy the currently displayed export data to the clipboard, ready to paste into your code editor.

Import

IMPORT FROM SPRITEPAD (OPEN SPD…)

Click **Open SPD…** to import sprite data from a SpritePad `.spd` file. SpritePad is a popular standalone C64 sprite editor and its file format is widely used in the C64 community. Imported sprites will populate the animation frames.



IMPORT FROM TEXT (IMPORT TEXT…)

Click **Import Text…** to open the text import window. Paste sprite data in any of the following formats and click **Import**:

- BASIC `DATA` statements
- Assembly `.byte` directives
- C arrays
- Raw hex bytes

If the pasted data contains multiple sprites worth of data, each sprite will be imported as a separate animation frame automatically. Use the **Replace existing frames** checkbox to control whether the imported data replaces your current frames or is added alongside them.

Animation

The Sprite Editor supports multi-frame animation, making it straightforward to design walking cycles, explosions, or any other animated sprite sequence.

THE ANIMATION STRIP

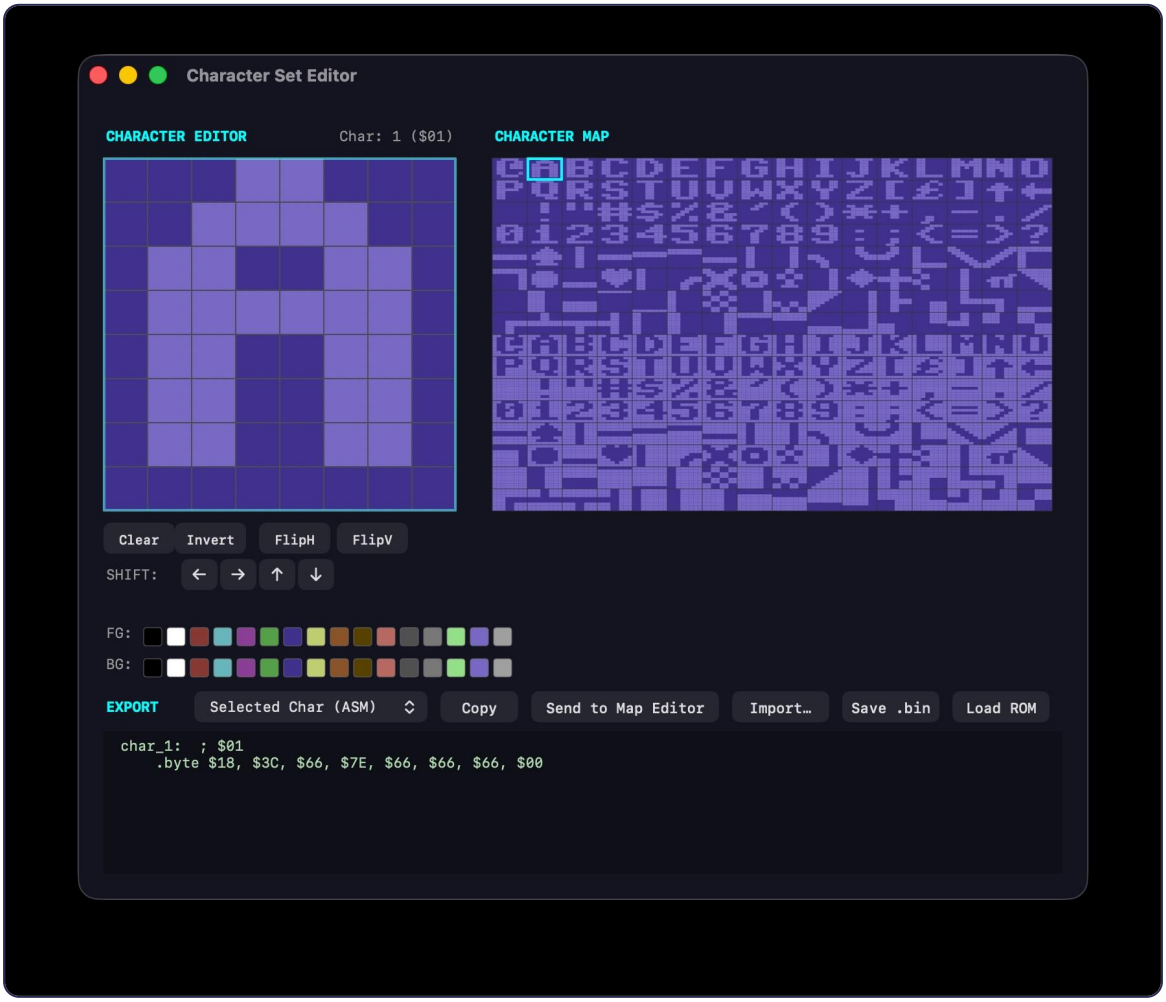
The animation strip along the bottom of the window shows thumbnails of all your animation frames. Click any thumbnail to select that frame for editing. Drag thumbnails left or right to reorder frames.

ANIMATION CONTROLS

CONTROL	DESCRIPTION
▶ Play	Plays the animation in the preview boxes at the current FPS setting.
Ping-Pong	When enabled, the animation plays forward through all frames then backward, looping continuously — useful for walk cycles and other symmetric animations.
FPS	Sets the playback speed in frames per second.
+ Add	Adds a new blank frame to the right of the currently selected frame.
Dup	Duplicates the currently selected frame and inserts the copy to its right. Useful for creating animation frames that are slight variations of each other.
Copy	Copies the currently selected frame's pixel data to the clipboard.
Paste	Pastes previously copied frame data into the currently selected frame.
Del	Deletes the currently selected frame.

ONION SKIN

Enable **Onion Skin** to see a ghost of the previous animation frame overlaid on the canvas while you draw the current frame. Use the **Opacity** slider to control how strongly the ghost image shows through. Onion skinning makes it much easier to draw smooth, consistent animation by keeping the previous frame visible as a reference.



Chapter 7: The Character Set Editor

The Character Set Editor lets you design custom character sets for your C64 programs. The C64 uses characters not just for text but as the building blocks of game backgrounds, UI elements, and graphics — so a custom character set is often central to a game's visual identity. The editor shows all 256 characters at once, lets you edit any of them freely, and links directly to the Game Map Editor so your map updates in real time as you work.

Open the Character Set Editor from **Tools → Character Set Editor** or by pressing ⌘⇧**D**.

The Interface

The Character Set Editor is divided into two main areas:

- **Left — Character Editor:** A large pixel grid where you edit the currently selected character.
- **Right — Character Map:** A live view of all 256 characters in your current character set.

The character map always reflects the current state of every character. Click any character in the map to load it into the editor on the left. The selected character is highlighted with a box in the map.

The current character number and its hex value are shown above the editor canvas — for example, `Char: 1 ($01)`.

Drawing

Click any pixel in the character editor canvas to toggle it on or off. Click and drag to draw continuously across multiple pixels without clicking each one individually. The C64 character format is 8×8 pixels.

All edits support full undo and redo (up to 10 levels) with ⌘**Z** and ⌘⇧**Z**.

The following operations are available below the canvas:

BUTTON	DESCRIPTION
Clear	Clears all pixels in the current character, leaving it blank.
Invert	Inverts all pixels in the current character — on pixels become off and vice versa.
Flip H	Flips the current character horizontally.

BUTTON	DESCRIPTION
Flip V	Flips the current character vertically.

Shift

The **Shift** arrows move all pixel data in the character one pixel in the chosen direction — up, down, left, or right. Pixels that shift off one edge of the canvas wrap around and reappear on the opposite edge. This is useful for nudging a character design into position or creating repeating patterns.

Colors

The **FG** (foreground) and **BG** (background) color rows let you customize the colors used to display characters in the editor and character map. Click any color swatch to apply it. These settings are for editing comfort only — they do not affect the exported character data itself. Set them to whatever colors make your designs easiest to see.

The Character Map

The character map on the right shows all 256 characters in your current character set, organized in a grid. It displays both the uppercase/graphics character set and the lowercase character set together.

The map updates in real time as you edit — every change you make to a character in the editor is immediately reflected in the map. This gives you a live overview of your entire character set as you work, so you can always see how individual characters look in context alongside their neighbors.

Linking to the Game Map Editor

Click **Send to Map Editor** to send your entire character set — all 256 characters — to the Game Map Editor for use as tiles. Once you've done this, the two editors become live-linked: any change you make to a character in the Character Set Editor is immediately reflected in the Game Map Editor, in real time. You can have both windows open side by side and watch your map update as you refine your characters — no need to re-export or refresh manually.

This live link is particularly powerful during game development, where you're often iterating on character designs while simultaneously building the map that uses them.

Import and Export

LOAD ROM

Click **Load ROM** to load the standard C64 English character set as your starting point. This is useful if you want to modify the built-in C64 font rather than designing from scratch, or if you want to restore the character set to its default state after experimenting.

IMPORT

Click **Import…** to load a raw character set binary file — for example, one previously saved with **Save .bin** or extracted from a C64 program or disk image. The imported data replaces the current character set.

SAVE .BIN

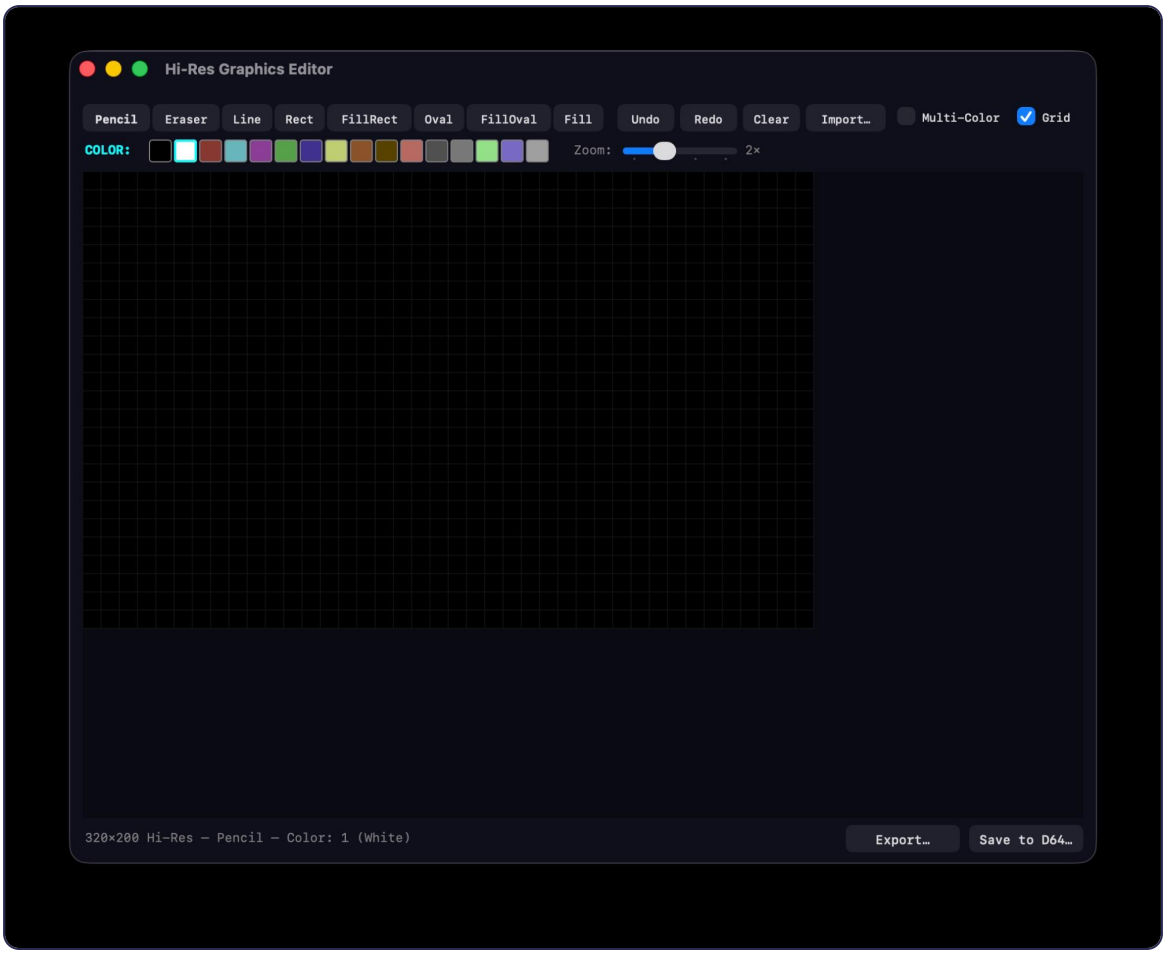
Click **Save .bin** to save your character set as a raw binary file. This format can be loaded directly by a C64 program using standard file I/O, or used as an import source for other tools.

EXPORT

The export area at the bottom of the window lets you export character data in several formats. Use the format dropdown to choose what to export and which characters to include:

FORMAT	SCOPE	DESCRIPTION
Selected Char (ASM)	Current character	Exports the selected character as ca65 `.byte` directives.
Selected Char (BASIC)	Current character	Exports the selected character as BASIC `DATA` statements.
Selected Char (Hex)	Current character	Exports the selected character as raw hex values.
Full Charset (ASM)	All 256 characters	Exports the entire character set as ca65 `.byte` directives.
Full Charset (BASIC)	All 256 characters	Exports the entire character set as BASIC `DATA` statements.

Click **Copy** to copy the export data to the clipboard, ready to paste into your code.



Chapter 8: The Hi-Res Graphics Editor

The Hi-Res Graphics Editor is a bitmap drawing tool designed around the C64's graphics modes. It gives you a full set of drawing tools, accurate color constraints, zoom support, and flexible export options — including a self-displaying PRG file that shows your artwork on a real C64 or in VICE without needing a separate loader program.

Open the Graphics Editor from **Tools → Graphics Editor** or by pressing **⌘⌥G**.

Graphics Modes

The Graphics Editor supports two C64 bitmap graphics modes:

HI-RES MODE (320×200)

The default mode. The canvas is 320×200 pixels — the full resolution of the C64's hi-res bitmap mode. In this mode each 8×8 cell can contain exactly 2 colors: a foreground and a background color. Different cells can use different color pairs, giving the full screen a wide range of colors overall despite the per-cell constraint.

MULTI-COLOR MODE (160×200)

Enable the **Multi-Color** checkbox to switch to the C64's multicolor bitmap mode. The canvas resolution drops to 160×200 pixels (each pixel is twice as wide on screen), and the color constraint changes: each 4×8 cell can contain up to 4 colors, with one global background color shared across the entire image. If you attempt to use more than 4 colors in a cell, the editor will automatically remap the colors in that cell to fit within the hardware constraint.

Multi-color mode is well suited to detailed illustrations and title screens where color variety matters more than sharp edges.

The Canvas

The drawing canvas fills the main area of the window. Use the **Zoom** slider in the toolbar to zoom in for detailed pixel work — zoom levels range from 1× up to 4×. Enable the **Grid** checkbox to overlay a pixel grid on the canvas, which is helpful when working at higher zoom levels.

The status bar at the bottom of the window shows the current canvas resolution, active tool, and selected color.

Drawing Tools

Select a tool from the toolbar to change the active drawing mode. All tools draw using the currently selected color.

TOOL	DESCRIPTION
Pencil	Draw individual pixels or drag to paint freehand.
Eraser	Erase pixels by dragging across them.
Line	Click and drag to draw a straight line between two points.
Rect	Click and drag to draw an unfilled rectangle.
FillRect	Click and drag to draw a filled rectangle.
Oval	Click and drag to draw an unfilled oval.
FillOval	Click and drag to draw a filled oval.
Fill	Flood-fills a contiguous area of the same color with the selected color.

Additional toolbar controls:

CONTROL	DESCRIPTION
Undo	Undoes the last edit (up to 10 levels).
Redo	Redoes a previously undone edit.
Clear	Clears the entire canvas.
Import…	Imports an existing graphics file.

All edits support full undo and redo (up to 10 levels) with ⌘**Z** and ⌘⇧**Z**.

Colors

The **COLOR** row in the toolbar shows all 16 C64 colors. Click any swatch to make it the active drawing color. The currently selected color is highlighted with a white border.

In multi-color mode, color constraints are enforced automatically per 4×8 cell. If adding a color to a cell would exceed the 4-color limit, the editor remaps the existing colors in that cell to accommodate the change.

Import

Click **Import…** to load an existing C64 graphics file. The editor accepts:

- **Art Studio format** — the native format of the popular C64 art package Micro Illustrator / Art Studio.

Export

Click **Export…** to save your artwork in one of the following formats:

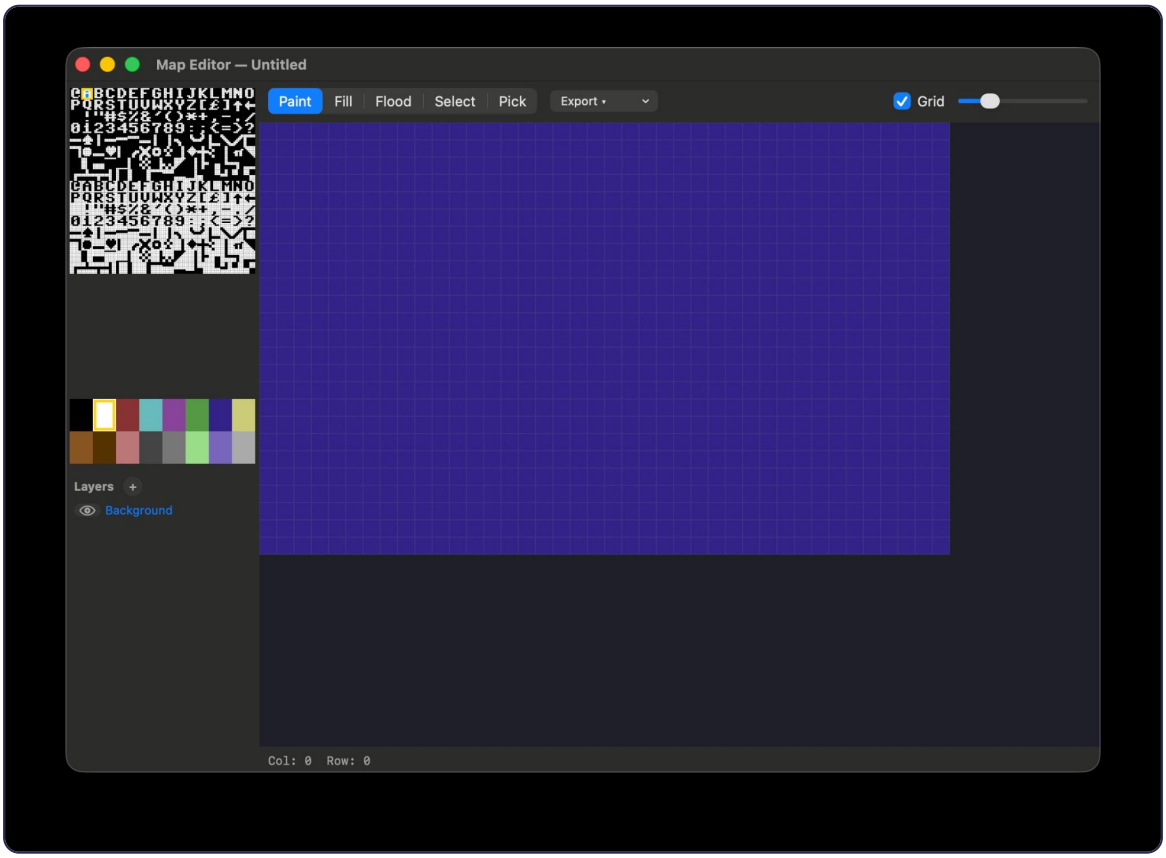
FORMAT	DESCRIPTION
Art Studio	Saves in Art Studio format, compatible with the original C64 application and other tools that support it.
Raw Binary	Saves the raw bitmap and color data as a binary file, suitable for loading directly into C64 memory.
Assembly	Exports the bitmap and color data as ca65 `.byte` directives, ready to include in an assembly source file.
BASIC Data	Exports the bitmap and color data as BASIC `DATA` statements.
Self-Displaying PRG	Creates a complete, standalone C64 program file that loads and displays your artwork when run. No separate loader required — just load and run on a real C64 or in VICE.

Save to D64

Click **Save to D64…** to save your artwork directly to a Commodore disk image. A dialog will ask whether to create a new D64 image or add to an existing one. You can save in any of the following formats:

- Art Studio
- Raw Binary
- Self-Displaying PRG

This makes it straightforward to prepare graphics files for distribution or for loading from disk in your C64 programs.



Chapter 9: The Game Map Editor

The Game Map Editor lets you build tile-based maps for your C64 games and programs using your custom character sets as tiles. It supports multiple layers, a full set of editing tools, and a live link to the Character Set Editor so your map updates in real time as you refine your characters.

Open the Game Map Editor from **Tools → Map Editor** (⌘⌥M), or create a new map with **File → New → New Map**.

The Map Canvas

The map canvas fills the main area of the window. It represents a 40×25 grid — exactly one full C64 text screen. Each cell in the grid corresponds to one character position on the C64 screen.

Enable the **Grid** toggle in the top right to show or hide the cell grid overlay on the canvas. The status bar at the bottom shows the current cursor position as **Col** and **Row** values.

The Character Palette

The character palette on the left side of the window shows your current character set — all 256 characters available as tiles. Click any character in the palette to select it as the active tile for painting.

The color swatches below the palette let you set the foreground and background colors used to display the characters in the palette and canvas, so you can preview your map in colors that match your intended design.

LIVE LINK WITH THE CHARACTER SET EDITOR

If you have sent a character set from the Character Set Editor using **Send to Map Editor**, the two windows become live-linked. Any change you make to a character in the Character Set Editor is immediately reflected in the map canvas — no need to re-import or refresh. See **Chapter 7: The Character Set Editor** for details.

Layers

The Game Map Editor supports multiple layers, giving you control over the visual depth of your map. The **Layers** panel on the left shows all current layers. Every new map starts with a single **Background** layer.

- Click the **+** button next to the Layers heading to add a new layer.
- Click the eye icon next to a layer to toggle its visibility.
- Click a layer name to make it the active editing layer.

You can have up to 16 layers in total, including the Background layer. Each layer is edited independently — painting on one layer does not affect any other.

Tools

Select a tool from the toolbar across the top of the window:

TOOL	DESCRIPTION
Paint	Click or drag to place the selected character tile on the active layer.
Fill	Fills a contiguous area of the same character with the selected tile.
Flood	Flood-fills the entire layer with the selected tile, regardless of existing content.
Select	Select one or more cells on the canvas. Selected cells can be copied and pasted, with full undo/redo support.
Pick	Click any character already placed on the canvas to make it the active tile instantly — much faster than finding it in the character palette.

> **Note:** The Pick tool is particularly useful during map editing sessions. Rather than hunting through the character palette to find a tile you've already placed somewhere on the map, just click it directly on the canvas to select it.

Export and Load

Click the **Export** button to access the following options:

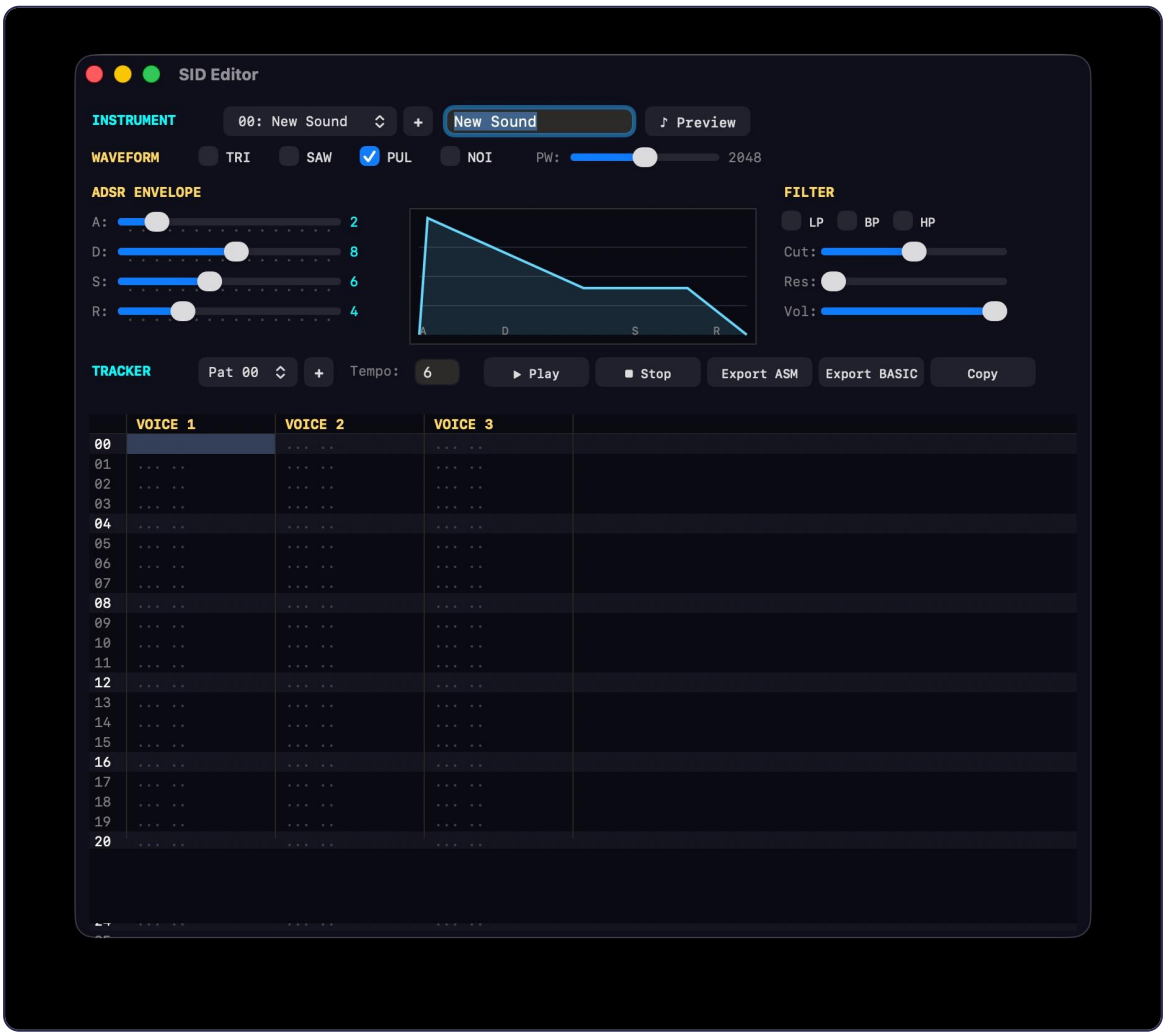
OPTION	DESCRIPTION
Assembly	Exports the map data as ca65 `.byte` directives, ready to include in an assembly source file.
Binary	Exports the map data as a raw binary file for direct loading into C64 memory.
Save as .c64map	Saves the map in C64 IDE's native map format, preserving all layers and settings for later editing.
Load Map	Loads a previously saved `.c64map` file.
Load Character Set	Loads a character set binary file to use as the tile palette for this map.

> **Note:** Load Map and Load Character Set are found in the Export menu — keep that in mind when you're looking for them.

Workflow Tips

- **Build your character set first.** The map editor is most powerful when used alongside the Character Set Editor. Design your tiles in the Character Set Editor, send them to the Map Editor, and then build your map with both windows open side by side.
- **Use Pick liberally.** When you're in the flow of placing tiles, Pick lets you grab any tile already on the map without breaking your rhythm to search the palette.
- **Use layers for foreground elements.** Put your background terrain on the Background layer and interactive or foreground elements on separate layers, making it easier to edit each independently.

- **Save your .c64map file often.** The native format preserves your full layer structure. Export to binary or assembly when you're ready to integrate the map into your program.



Chapter 10: The SID Editor

The SID Editor is a tracker-style music composition tool built around the C64's SID (Sound Interface Device) chip. It combines an instrument designer with a pattern-based tracker for all three SID voices, and can export your music directly as BASIC DATA statements or 6502 assembly ready to drop into your program.

Open the SID Editor from **Tools → SID Editor** or by pressing ⌘⇧**M**.

The SID Chip

The C64's SID chip is one of the most celebrated sound chips in computing history. It provides three independent voices, each capable of producing sound with a programmable waveform, ADSR amplitude envelope, and a shared filter. The SID Editor gives you direct access to all of these capabilities in a familiar tracker interface.

Instruments

The **INSTRUMENT** section at the top of the window lets you create and manage the sounds used in your music.

- Use the dropdown to select the active instrument. Instruments are numbered from 00 upward.
- Click **+** to create a new instrument.
- Type a name for the instrument in the name field to the right of the dropdown.
- Click ♩ **Preview** to play a brief test note using the current instrument settings, so you can hear how it sounds without entering it into the tracker.

You can create as many instruments as your music requires.

Waveform

The **WAVEFORM** section selects the type of sound wave the SID chip generates for this instrument. The C64's SID supports four waveform types, and you can combine multiple waveforms simultaneously for more complex timbres:

WAVEFORM	DESCRIPTION
TRI	Triangle wave — a soft, hollow tone.
SAW	Sawtooth wave — a bright, buzzy tone, good for bass lines and leads.

WAVEFORM	DESCRIPTION
PUL	Pulse wave — a versatile tone whose character changes dramatically with the pulse width setting.
NOI	Noise — random noise, useful for percussion and sound effects.

The **PW** (Pulse Width) slider is only active when the PUL waveform is selected. It controls the duty cycle of the pulse wave, ranging from a thin, reedy sound at low values to a full, hollow square wave at the midpoint.

ADSR Envelope

The **ADSR ENVELOPE** section shapes how the volume of the instrument changes over time:

PARAMETER	DESCRIPTION
A (Attack)	How quickly the sound rises to full volume after a note is triggered.
D (Decay)	How quickly the sound falls from full volume to the sustain level.
S (Sustain)	The volume level held while the note is held.
R (Release)	How quickly the sound fades to silence after the note is released.

The envelope graph in the center displays the current ADSR shape visually. It is a display only — adjust the values using the sliders on the left.

Filter

The **FILTER** section applies the SID chip's resonant filter to the instrument. You can combine multiple filter types simultaneously:

FILTER	DESCRIPTION
LP	Low-pass — allows low frequencies through, cuts high frequencies. Produces a warm, muffled tone.
BP	Band-pass — allows a band of frequencies through, cuts both high and low.
HP	High-pass — allows high frequencies through, cuts low frequencies. Produces a thin, bright tone.

CONTROL	DESCRIPTION
Cut	Cutoff frequency — sets the point at which the filter takes effect.
Res	Resonance — boosts frequencies around the cutoff point, adding a characteristic peak or "wah" quality.

The **Vol** slider controls the overall global output volume for the SID chip. This is a single global setting that affects all three voices simultaneously, reflecting the SID chip's hardware architecture.

The Tracker

The **TRACKER** section is where you compose music by entering notes into a pattern grid. The tracker has three columns — one for each of the SID chip's three voices — and 32 rows per pattern.

PATTERNS

Use the **Pat** dropdown to select the active pattern. Click **+** to add a new pattern. You can create as many patterns as your music requires. The tracker plays one pattern at a time, but all patterns are included when you export.

SPEED

The **Speed** field controls the playback rate. This follows the SID tracker convention where a lower number means faster playback and a higher number means slower playback. This is the opposite of BPM-based tempo as used in most music software, but is standard in the C64 SID tracker world.

ENTERING NOTES

Click any cell in the tracker grid to make it active. Then use the keyboard to enter notes using the standard tracker keyboard layout, where two rows of keys map to a piano-style chromatic scale:

Upper row (higher octave):

KEY	NOTE
Q	C
2	C#
W	D
3	D#
E	E
R	F
5	F#
T	G
6	G#
Y	A
7	A#
U	B

Lower row (lower octave):

KEY	NOTE
Z	C
S	C#
X	D
D	D#

KEY	NOTE
C	E
V	F
G	F#
B	G
H	G#
N	A
J	A#
M	B

After entering a note, the active cell automatically advances to the next row, so you can enter a melody quickly without manually moving the cursor.

To stop a note playing — entering a rest or note-off — type **.** (period) in any cell. Without a note-off, the SID chip will continue playing the last note until a new note is entered or playback stops.

> **Note:** Octave selection is not yet supported — all notes are entered at a fixed octave. Octave control is planned for a future update.

PLAYBACK CONTROLS

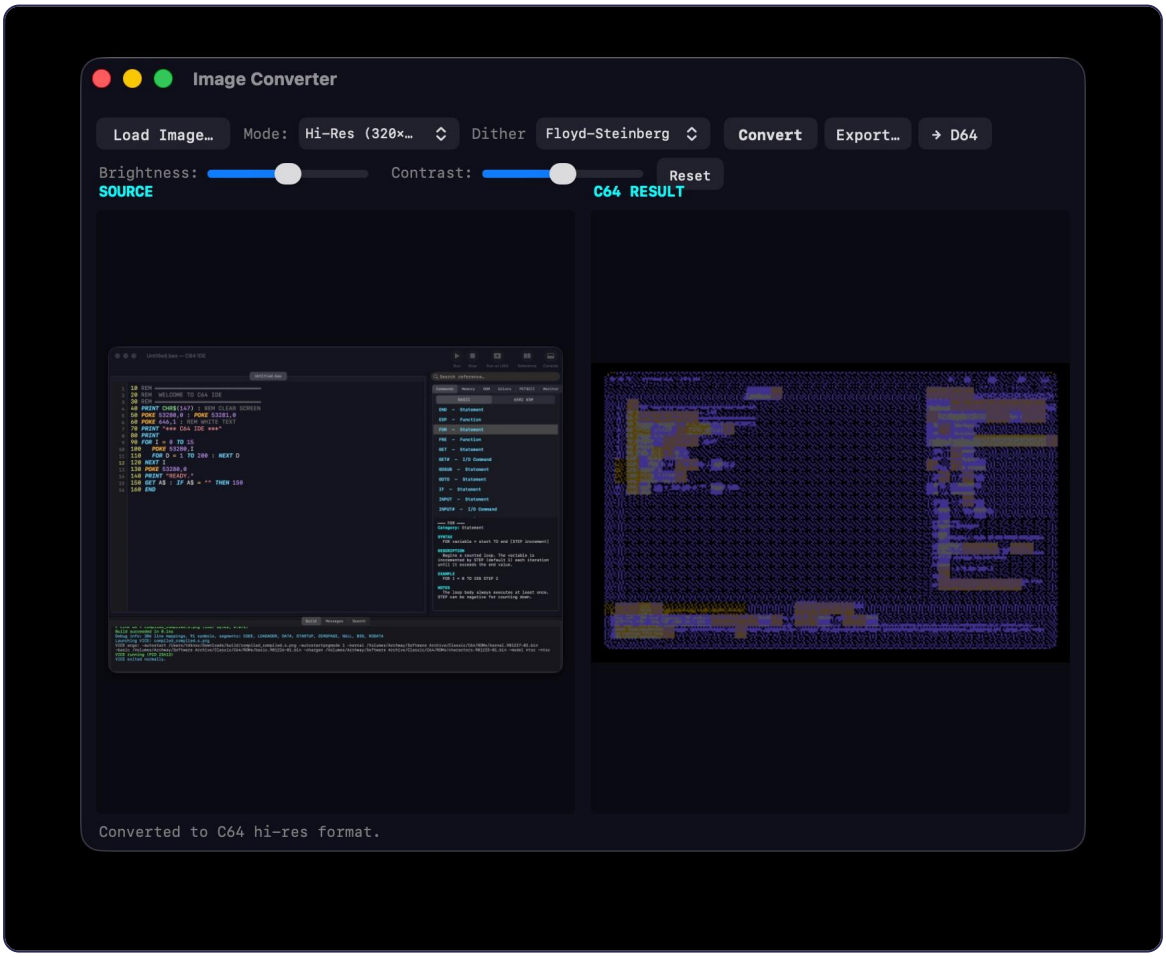
CONTROL	DESCRIPTION
▶ **Play**	Plays the current pattern from the beginning.
■ **Stop**	Stops playback.

Export

Click **Export ASM** to export all patterns and instrument data as ca65-compatible 6502 assembly source code, ready to include in your program.

Click **Export BASIC** to export as BASIC DATA statements.

Click **Copy** to copy the export data to the clipboard.



Chapter 11: The Image Converter

The Image Converter lets you convert modern photographs and artwork into C64-compatible bitmap graphics. It handles the conversion from full-color images to the C64's limited palette automatically, with dithering options and real-time brightness and contrast controls to help you get the best possible result.

Open the Image Converter from **Tools → Image Converter** or by pressing **⌘⌥I**.

Loading an Image

Click **Load Image…** to open any image file supported by macOS — including JPEG, PNG, TIFF, and most other common image formats. The original image will appear on the left side of the window as the source.

Conversion Settings

MODE

Use the **Mode** dropdown to select the target C64 graphics mode:

MODE	DESCRIPTION
Hi-Res (320×200)	Converts to the C64's hi-res bitmap mode. Each 8×8 cell can contain 2 colors. Produces sharper edges but fewer colors per area.
Multi-Color (160×200)	Converts to the C64's multicolor bitmap mode. Each 4×8 cell can contain up to 4 colors, with one shared background color. Produces more color variety at the cost of horizontal resolution.

DITHERING

Dithering spreads quantization error across neighboring pixels to simulate colors that the C64's palette can't represent exactly. Use the **Dither** dropdown to choose a method:

METHOD	DESCRIPTION
Floyd-Steinberg	Error-diffusion dithering that produces natural-looking results with smooth gradients. Generally the best choice for photographs.
Ordered (Bayer)	

METHOD	DESCRIPTION
	Pattern-based dithering that produces a regular, structured appearance. Can look good for certain types of artwork and gives a more retro aesthetic.
None	No dithering — each pixel is simply mapped to the nearest C64 color. Produces the most accurate color mapping but can look posterized on images with gradients.

Converting

Click **Convert** to perform the conversion. The C64 result will appear on the right side of the window.

Brightness and Contrast

After converting, use the **Brightness** and **Contrast** sliders to fine-tune the result. These controls are applied in real time — the image is fully reconverted with dithering recalculated every time you adjust a slider, giving you an accurate preview of the final result rather than just a simple filter overlay.

Click **Reset** to return both sliders to their default positions.

The typical workflow is:

1. Load your image.
2. Choose a mode and dithering method.
3. Click **Convert**.
4. Adjust Brightness and Contrast until the result looks right.
5. Export or save to D64.

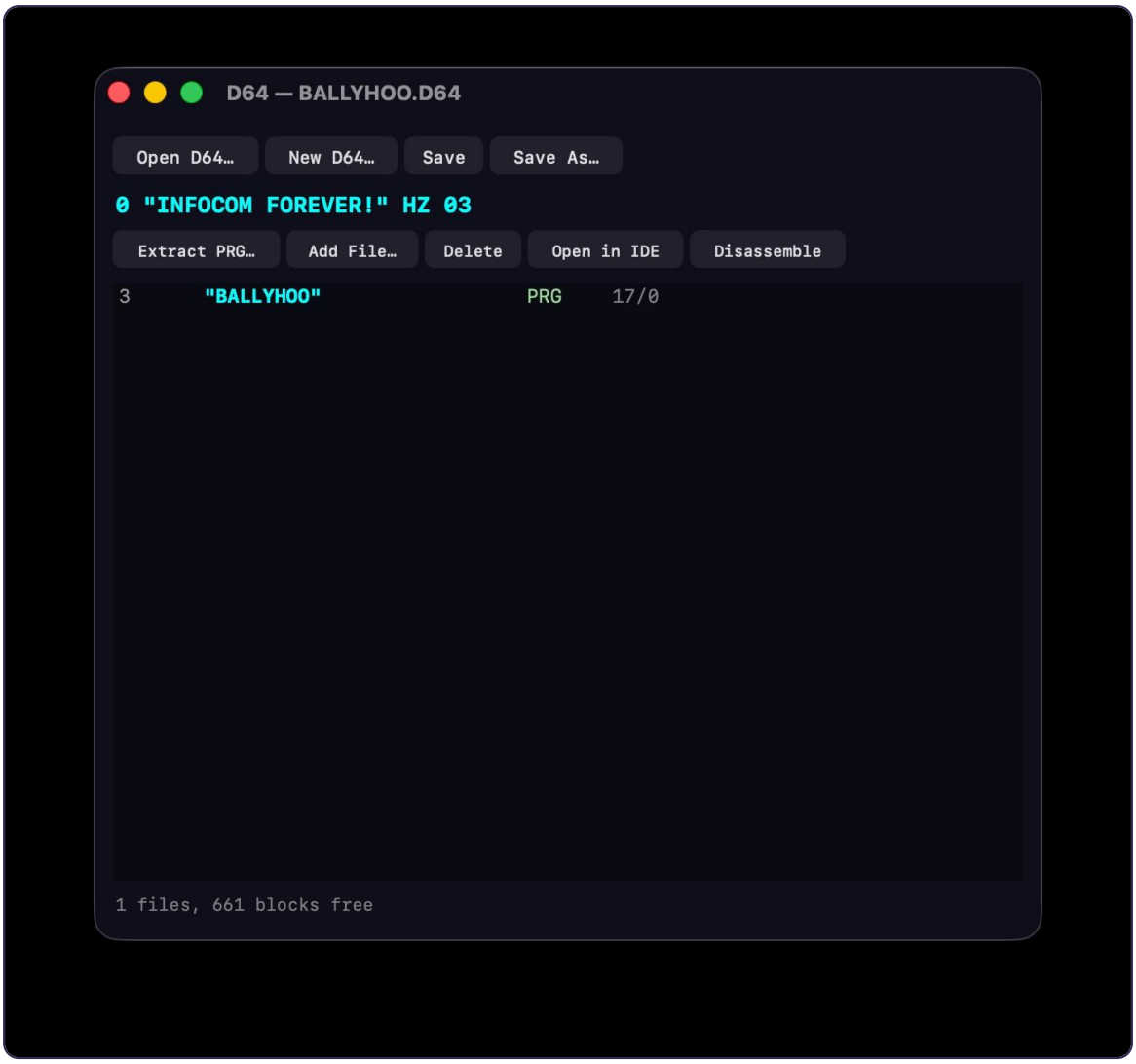
Export

Click **Export...** to save the converted image in one of the following formats:

FORMAT	DESCRIPTION
Art Studio	Saves in Art Studio format, compatible with the original C64 application and other tools that support it.
Assembly	Exports the bitmap and color data as ca65 `.byte` directives for use in an assembly source file.
BASIC Data	Exports the bitmap and color data as BASIC `DATA` statements.
Self-Displaying PRG	Creates a standalone C64 program that loads and displays the image when run. No separate loader required.

Save to D64

Click **→ D64** to save the converted image directly to a Commodore disk image as a self-displaying PRG file. A dialog will ask whether to create a new D64 image or add to an existing one. Load the resulting disk image in VICE or on real hardware and run the file to see your converted image displayed on the C64.



Chapter 12: The D64 Disk Browser

The D64 Disk Browser lets you open, create, and manage Commodore 1541 disk images directly in C64 IDE. You can add and extract files, open programs directly in the editor, and send files to the disassembler — all without leaving the app.

Open the D64 Disk Browser from **Tools → D64 Disk Browser** or by pressing ⌘⇧**K**.

Opening and Creating Disk Images

BUTTON	DESCRIPTION
Open D64…	Opens an existing `.d64` disk image file.
New D64…	Creates a new, blank `.d64` disk image.
Save	Saves changes to the current disk image.
Save As…	Saves the current disk image to a new file.

The Disk Directory

The directory listing shows the contents of the currently open disk image, in the same format as a real Commodore 1541 floppy disk.

The header line shows the disk name and ID as they were set when the disk image was formatted — for example:

```
0 "MY DISK" AB 01
```

This information reflects what was written when the disk image was created and is not editable.

Each file in the directory is shown with four pieces of information:

COLUMN	DESCRIPTION
Blocks	The number of 254-byte blocks the file occupies on the disk.
Filename	The name of the file as stored on the disk, in PETSCII.
Type	The file type — typically `PRG` for program files.
Track/Sector	The track and sector number of the first block of the file on the disk.

The status bar at the bottom of the window shows the total number of files and the number of blocks remaining free on the disk.

Working with Files

Select a file in the directory listing to make it active, then use the toolbar buttons to work with it:

EXTRACT PRG…

Extracts the selected file from the disk image and saves it as a regular file on your Mac. A save dialog will prompt you for the destination.

ADD FILE…

Adds a file from your Mac to the disk image. C64 IDE supports adding the following file types:

- `.bas` — Commodore BASIC source files
- `.s` — 6502 assembly source files
- `.prg` — Compiled C64 program files

DELETE

Deletes the selected file from the disk image. This cannot be undone, so make sure you have a copy elsewhere if you need it.

OPEN IN IDE

Opens the selected file in the Code Editor:

- **BASIC programs** — opened directly as an editable BASIC source file in a new editor tab.
- **Machine language programs** — C64 IDE will offer to disassemble the file automatically. If you accept, the disassembled code opens in a new editor tab, ready to browse or edit.

DISASSEMBLE

Runs the selected file through the 6502 Disassembler and opens the result in a new editor tab. See **Chapter 13: The 6502 Disassembler** for details on the disassembly output.

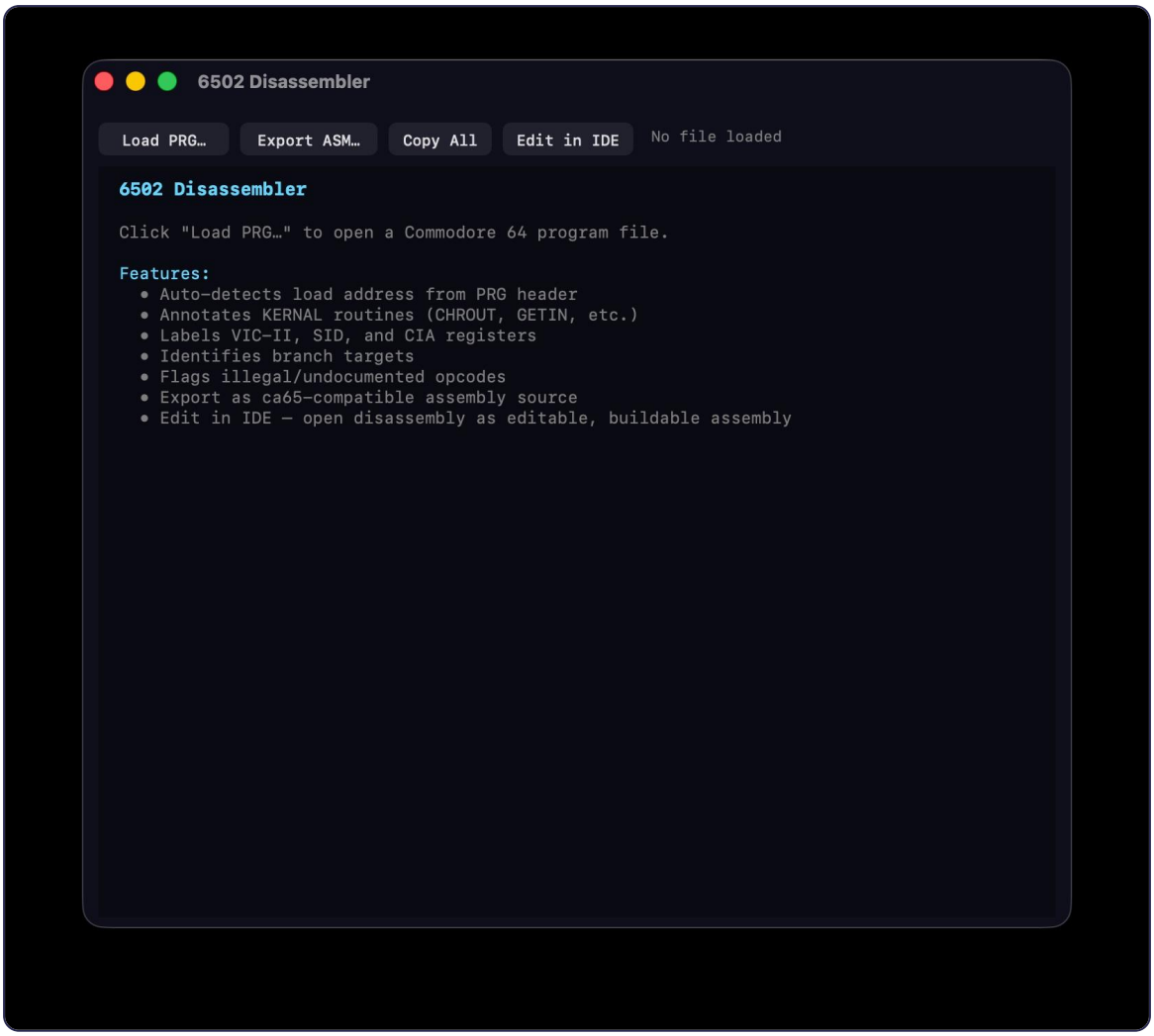
Limitations

The D64 Disk Browser reflects the real-world constraints of the Commodore 1541 disk format:

- The directory is flat — subdirectories are not supported, just as on a real 1541.
- A standard D64 image holds a maximum of 144 files and 664 blocks (approximately 170KB) of data.

- #### RENAME

Double-click any filename in the directory listing to rename it. The new name must conform to Commodore 1541 filename rules — up to 16 characters, PETSCII character set.



Chapter 13: The 6502 Disassembler

The 6502 Disassembler takes any Commodore 64 PRG file and converts it back into readable 6502 assembly language. What sets it apart from a basic disassembler is its built-in knowledge of the C64's memory map — it automatically replaces raw addresses with meaningful names for KERNAL routines, VIC-II registers, SID registers, CIA registers, and other well-known memory locations, making disassembled code dramatically easier to understand.

Open the Disassembler from **Tools → Disassembler** or by pressing ⌘⇧**I**.

Loading a Program

Click **Load PRG…** to open any Commodore 64 PRG file. The disassembler reads the two-byte load address from the PRG header automatically — no need to specify the address manually.

The disassembly output appears in the main text area. You can also reach the disassembler directly from the D64 Disk Browser by selecting a file and clicking **Disassemble**.

The Disassembly Output

The disassembler produces a clean assembly listing showing the address, mnemonic, operand, and any relevant annotations for each instruction.

AUTOMATIC SYMBOL RESOLUTION

The disassembler checks every address reference against its built-in C64 symbol tables and substitutes descriptive names wherever it can. For example:

- `JSR $FFD2` becomes `JSR CHROUT`
- `STA $D020` becomes `STA VIC_BORDERCOLOR`
- `LDA $DC01` becomes `LDA CIA1_PORTB`

This applies to KERNAL routine addresses, VIC-II registers, SID registers, CIA registers, and other known C64 memory locations. The result is a disassembly that reads much more like human-written assembly code than a raw hex dump.

BRANCH TARGETS

Branch instructions (`BEQ`, `BNE`, `BCC`, `BCS`, etc.) have their target addresses calculated and displayed, making it easy to follow the flow of control through the code.

ILLEGAL AND UNDOCUMENTED OPCODES

The C64's 6510 processor has a number of illegal opcodes — byte values that are not defined in the official 6502 instruction set but produce consistent

behavior on real hardware. Some C64 programs use these deliberately for size or speed optimization. The disassembler handles them gracefully:

```
C000  02  ???  ; ILLEGAL OPCODE
```

Illegal opcodes are displayed with their address and hex value, flagged clearly with `; ILLEGAL OPCODE` so you know not to treat them as standard instructions.

Toolbar Actions

BUTTON	DESCRIPTION
Load PRG…	Opens a PRG file for disassembly.
Export ASM…	Saves the disassembly output to a `.s` file on disk.
Copy All	Copies the entire disassembly output to the clipboard.
Edit in IDE	Opens the disassembly as a new assembly source file in the Code Editor, ready to browse, edit, and rebuild.

Edit in IDE

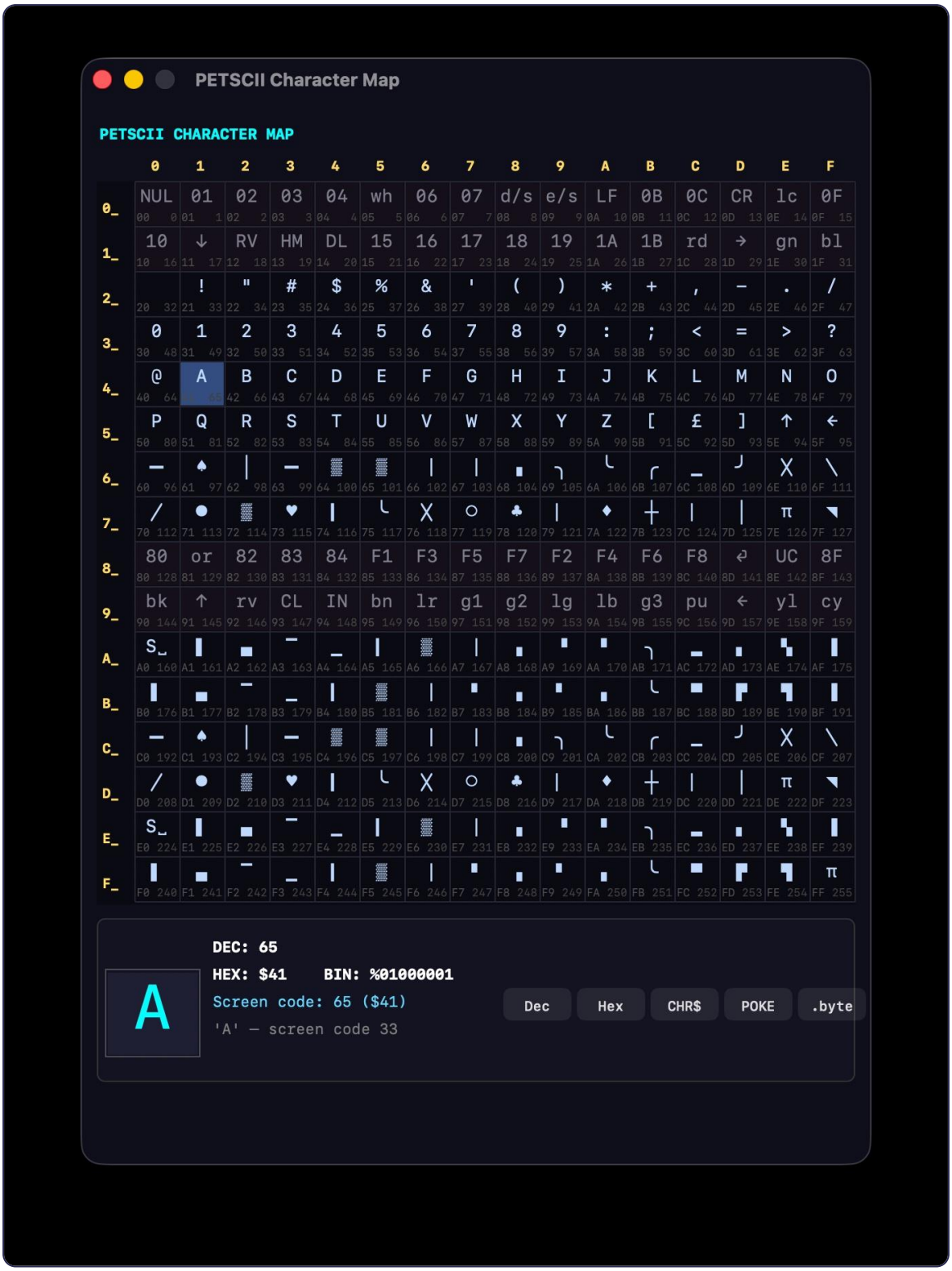
The **Edit in IDE** button is particularly powerful. It takes the disassembly output and opens it as a fully editable assembly file in a new Code Editor tab. From there you can:

- Browse and study the code with full syntax highlighting and the Reference panel alongside.
- Make changes and rebuild the program with ⌘**R**.
- Use the VICE Debugger to step through the code.

This makes it straightforward to reverse-engineer, study, or modify existing C64 programs — load a PRG, disassemble it, open it in the IDE, and start exploring.

Limitations

- The disassembler has no way of distinguishing code from data — it disassembles everything as instructions. If a program stores data inline with code, those bytes will be disassembled as instructions, which may produce nonsensical output in those regions.
- There is currently no way to provide hints to the disassembler, such as marking a region as data rather than code.
- Self-modifying code may not disassemble meaningfully, since the disassembler works on the static file rather than observing the program at runtime.



Chapter 14: The PETSCII Character Map

The PETSCII Character Map is a full interactive reference for the C64's PETSCII character set. PETSCII (PET Standard Code of Information Interchange) is the character encoding used by all Commodore 8-bit computers — similar in

concept to ASCII but with a completely different character layout, including the C64's distinctive graphic block characters and control codes.

Open the PETSCII Character Map from **Tools → PETSCII Map** or by pressing ⌘⇧**P**.

The Character Grid

The main area of the window displays all 256 PETSCII characters in a 16×16 grid, organized by code value. The column headers show the low nibble (0–F) and the row headers show the high nibble (0_–F_), so any character's PETSCII code can be read directly from its position in the grid.

All 256 characters are visible at once — the map does not scroll.

Click any character to select it and view its full details in the panel at the bottom of the window.

The Detail Panel

Clicking a character displays the following information in the detail panel:

- **A large preview** of the character as it appears on the C64.
- **DEC** — the decimal PETSCII code value.
- **HEX** — the hexadecimal PETSCII code value (prefixed with `$`).
- **BIN** — the binary PETSCII code value (prefixed with `%`).
- **Screen Code** — the corresponding C64 screen code value, which is different from the PETSCII code and is what you write directly to screen memory at `$0400`.
- **A plain English description** of the character — for example "Circle Outline", "Brown (color)", or "Function Key F2". This is especially useful for control codes and graphic characters that have no obvious visual representation.

Copying Values

Five buttons in the detail panel let you copy the selected character's value to the clipboard in the format most useful for your current task:

BUTTON	COPIES
Dec	The decimal PETSCII value — e.g. `65`
Hex	The hexadecimal value — e.g. `$41`
CHR$	A BASIC `CHR$()` expression — e.g. `CHR$(65)`
POKE	A BASIC `POKE` expression for writing to screen memory — e.g. `POKE 1024,65`
.byte	A ca65 assembly `.byte` directive — e.g. `.byte $41`

PETSCII vs Screen Codes

One of the most common points of confusion in C64 programming is the difference between PETSCII codes and screen codes. They are two different ways of referring to characters:

- **PETSCII codes** are used with BASIC commands like `PRINT CHR$(n)` and with KERNAL routines like `CHROUT`. The KERNAL handles the translation to screen memory for you.
- **Screen codes** are what you write directly to the C64's screen RAM at `$0400–$07E7` when doing fast direct screen writes in assembly. They are a different mapping of the same 256 characters.

The detail panel shows both values for every character, so you always have the right number to hand whichever approach you're using.

Chapter 15: The C64 Character ROM Viewer

The C64 Character ROM Viewer is a read-only reference tool for browsing the characters built into the C64's character ROM. It shows all 256 characters from both of the C64's built-in character sets, with full pixel data, code values, and ready-to-use POKE examples for each one.

Open the Character ROM Viewer from **Tools → ROM Character Set Viewer**.

The Two Character Sets

The C64's character ROM contains two complete character sets, selectable with the tabs at the top of the window:

TAB	DESCRIPTION
Set 1: Upper/ Graphics	The default character set. Contains uppercase letters, digits, punctuation, and the C64's distinctive graphic block characters. This is the set active when the C64 first boots.
Set 2: Lower/Upper	The alternate character set. Contains both lowercase and uppercase letters, making it suitable for text-heavy programs. Activated by pressing **Shift+Commodore** on a real C64.

Browsing Characters

The main area of the window displays all 256 characters in the selected set as a grid of 8×8 pixel tiles. Click any character to select it and view its full details in the scrollable panel below.

The Detail Panel

The detail panel shows comprehensive information about the selected character:

- **A large 8×8 pixel grid** showing the character's exact pixel layout.
- **Screen Code** — the value to write directly to screen RAM at `$0400` to display this character.
- **PETSCII Code** — the PETSCII value used with BASIC's `CHR$()` or the KERNAL's `CHROUT` routine.
- **ROM Offset** — the byte offset of this character's data within the character ROM, and which set it belongs to.
- **Name** — a plain English name for the character, for example `Graphics 116` or `Latin Capital Letter A`.
- **Pixel Data** — the eight bytes that make up the character, shown as a visual dot/block representation alongside the hex value for each row. For example:

```
── Character 116 ──
Screen Code:  $74 (116)
PETSCII Code: $B4 (180)
ROM Offset:   $03A0 (Set 1)

Name: Graphics 116

Pixel Data:
  ■····· ($C0)
  ■····· ($C0)
  ■····· ($C0)
  ■····· ($C0)
  ■····· ($C0)
  ■····· ($C0)
  ■····· ($C0)
  ■····· ($C0)

POKE to screen:
  POKE 1024,116
  POKE 55296,1  (white)

Reversed:     Screen code $F4 (244)
```

- **POKE to screen** — ready-to-use BASIC `POKE` statements to place this character at the first screen position (`$0400`) in white. Adapt the address and color value as needed for your program.
- **Reversed** — the screen code for the reversed (inverted) version of this character, where the foreground and background pixels are swapped.

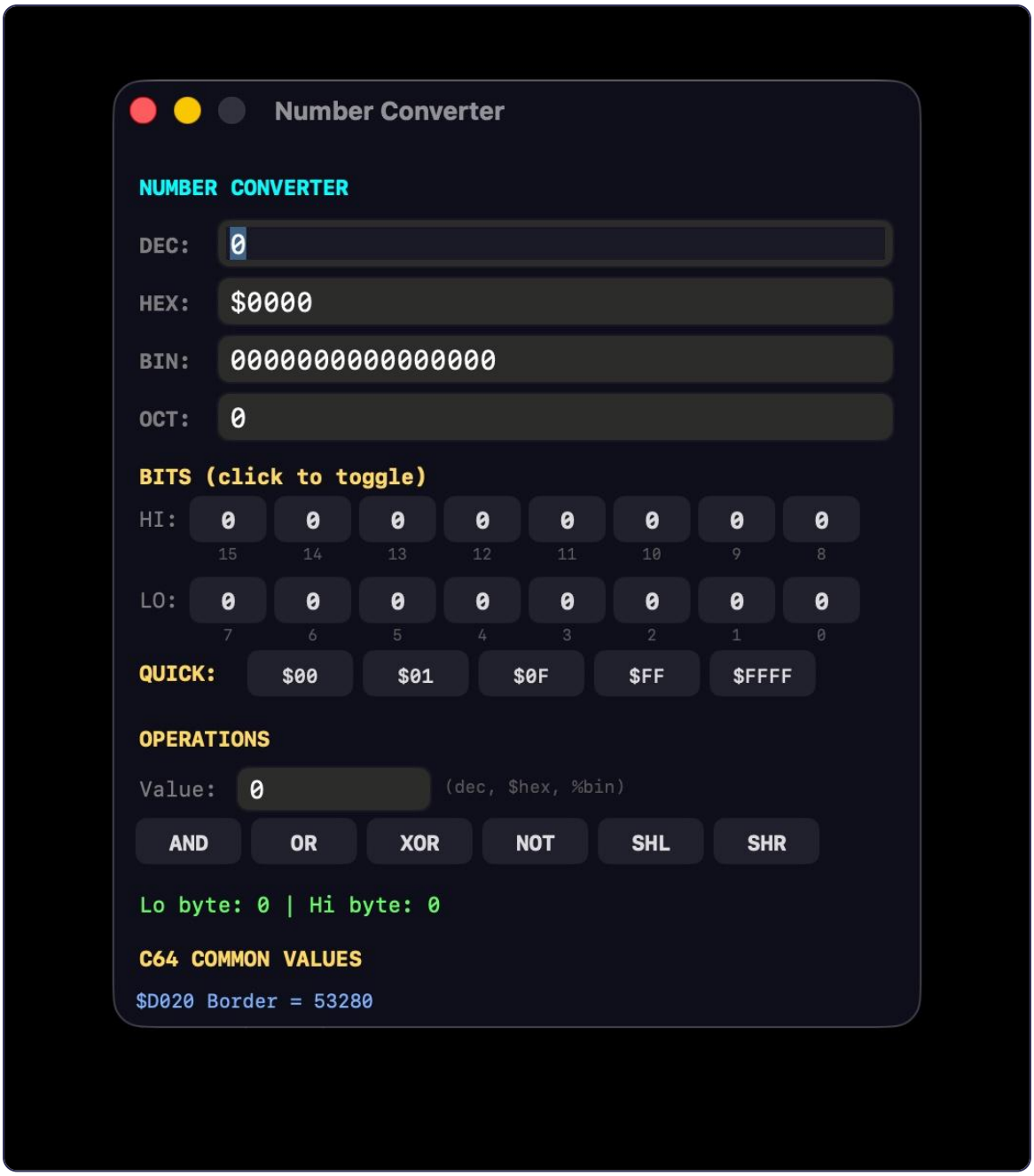
The detail panel is scrollable to accommodate the full pixel data display.

Usage

The Character ROM Viewer is a reference tool only — the character data cannot be edited here. To design custom characters based on the ROM set as

a starting point, use the Character Set Editor (**Chapter 7**) and click **Load ROM** to load the standard character set for editing.

Use this viewer when you need to quickly look up the screen code, PETSCII value, or pixel layout of a specific built-in character while working on your program.



Chapter 16: The Number Converter

The Number Converter is a programmer's calculator tailored for C64 development. It converts values between decimal, hexadecimal, binary, and octal simultaneously, lets you toggle individual bits, and supports common bitwise operations — all in the number formats C64 programmers use every day.

Open the Number Converter from **Tools → Number Converter** or by pressing ⌘⇧**U**.

Number Display

The top section displays the current value in four formats simultaneously:

FIELD	FORMAT	EXAMPLE
DEC	Decimal	`65`
HEX	Hexadecimal	`$0041`
BIN	Binary	`0000000001000001`
OCT	Octal	`101`

Type a value into any field and press **Return** or **Tab** to update all other fields instantly. Notes on input format:

- **HEX** — the `$` prefix is always shown and cannot be removed. Type your hex digits after it.
- **BIN** — no `%` prefix required. Just type the binary digits directly.
- **DEC** and **OCT** — type the number directly.

Bit Toggles

The **BITS** section displays the current value as 16 individual bit buttons, arranged in two rows:

- **HI** — bits 15 down to 8 (the high byte)
- **LO** — bits 7 down to 0 (the low byte)

Click any bit button to toggle it between 0 and 1. All four number display fields update immediately to reflect the new value. This is particularly useful when working with C64 hardware registers where individual bits control

specific features — for example the VIC-II control registers or the CIA port direction registers.

The **Lo byte** and **Hi byte** values are also displayed in decimal below the bit toggles, useful when you need to split a 16-bit address into its component bytes for assembly code.

Quick Values

The **QUICK** buttons load commonly used values instantly:

BUTTON	VALUE
$00	0 — all bits off
$01	1 — bit 0 set
$0F	15 — low nibble all on
$FF	255 — all bits on (full byte)
$FFFF	65535 — all bits on (full word)

Bitwise Operations

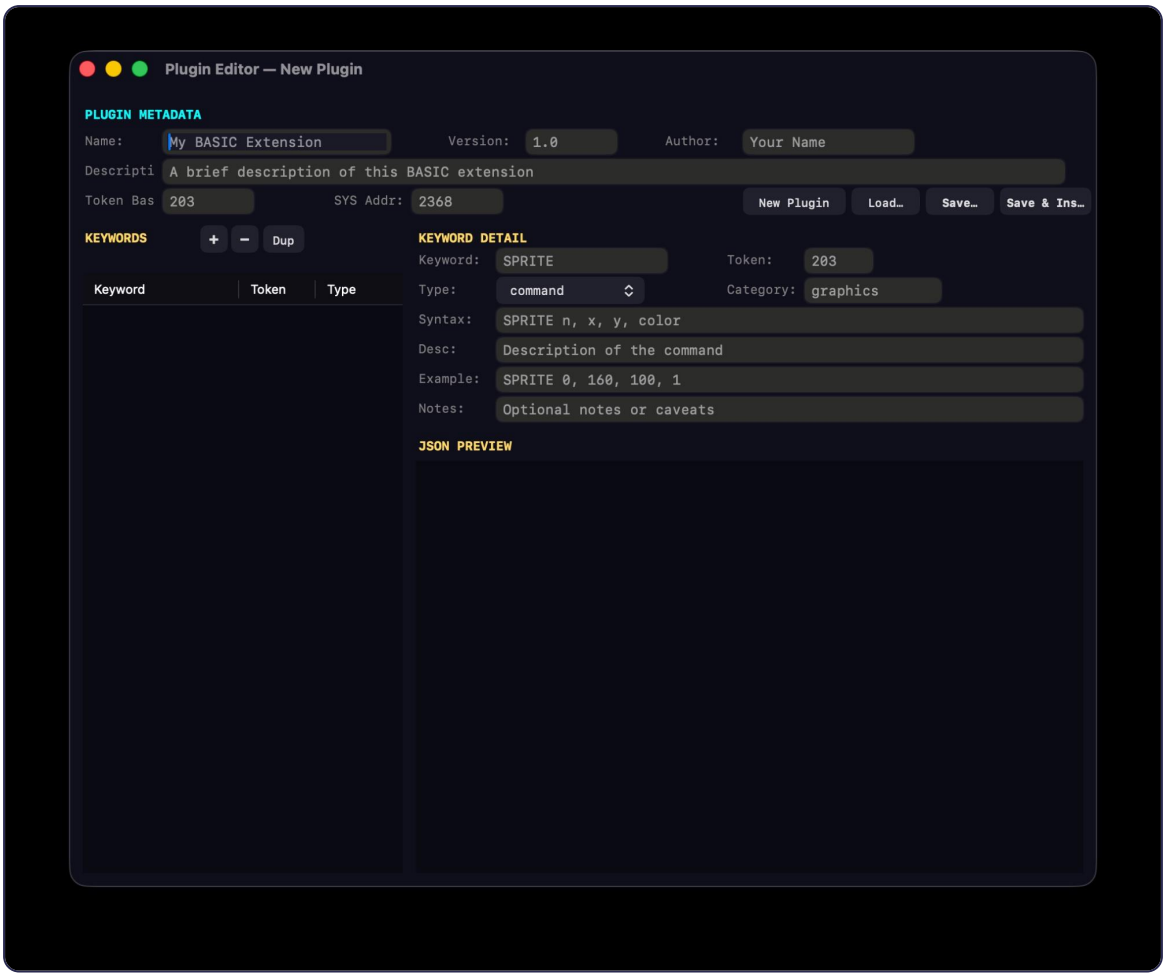
The **OPERATIONS** section lets you apply a bitwise operation to the current value. Enter a second value in the **Value** field using any of the following formats:

- Decimal — e.g. `128`
- Hexadecimal — e.g. `$80`
- Binary — e.g. `%10000000`

Then click the operation to apply:

OPERATION	DESCRIPTION
AND	Bitwise AND — keeps only the bits that are set in both values. Useful for masking.
OR	Bitwise OR — sets any bit that is set in either value. Useful for combining flags.
XOR	Bitwise exclusive OR — sets bits that differ between the two values. Useful for toggling specific bits.
NOT	Bitwise NOT — inverts all bits in the current value. The second value field is ignored.
SHL	Shift left — shifts the current value left by the number of bits specified in the value field. Equivalent to multiplying by a power of 2.
SHR	Shift right — shifts the current value right by the number of bits specified in the value field. Equivalent to dividing by a power of 2.

The result replaces the current value and all display fields update immediately.



Chapter 17: The BASIC Keyword/Plugin Editor

C64 IDE's plugin system lets you extend the IDE to support enhanced BASIC interpreters beyond standard Commodore BASIC V2. A plugin teaches C64 IDE about a custom BASIC dialect's keywords — their token values, syntax, and descriptions — so the syntax highlighter, tokenizer, detokenizer, and hover tooltips all work correctly for that dialect. VisionBASIC 1.1 is included as a ready-to-use example plugin.

BASIC Dialects

Access the plugin system from **Tools → BASIC Dialect**, which opens a submenu with the following options:

OPTION	DESCRIPTION
Standard BASIC	Use standard Commodore BASIC V2 — the built-in BASIC of the C64.
VisionBASIC	Use VisionBASIC 1.1, an enhanced BASIC dialect with graphics, sound, sprite, interrupt, and inline assembly commands. Included as a pre-built example plugin.
Install Plugin…	Install a plugin file (`.c64basic`) provided by a third party. The file is copied to the Plugins folder and activated immediately.
Edit Plugin…	Open the Plugin Editor to create a new plugin or modify an existing one.
Reveal Plugins Folder	Opens the folder where C64 IDE stores plugin files in the Finder. Any `.c64basic` file placed in this folder will be loaded automatically on the next launch and added to the BASIC Dialect submenu.

The active dialect affects syntax highlighting, tokenization, detokenization, and keyword tooltips throughout the IDE.

The Plugin Editor

Open the Plugin Editor via **Tools → BASIC Dialect → Edit Plugin…**. This is where you create or modify plugin files.

PLUGIN METADATA

The top section defines the overall plugin:

FIELD	DESCRIPTION
Name	The name of the BASIC dialect, as it will appear in the BASIC Dialect submenu.
Version	The version number of the plugin.
Author	The plugin author's name.
Description	A brief description of the dialect and what it adds.

FIELD	DESCRIPTION
Token Base	The token value where this dialect's extended keywords begin. Standard C64 BASIC V2 uses tokens up to $CB — most extensions start at $CB or higher.
SYS Addr	The address to `SYS` to activate the BASIC extension before running a program.

Use **New Plugin** to start a fresh plugin, **Load…** to open an existing `.c64basic` file for editing, **Save…** to save the plugin to disk, and **Save & Install** to save and immediately activate the plugin in the IDE.

MANAGING KEYWORDS

The **KEYWORDS** panel on the left lists all keywords defined in the plugin. Use the buttons above the list to manage them:

BUTTON	DESCRIPTION
+	Add a new blank keyword entry.
−	Delete the selected keyword.
Dup	Duplicate the selected keyword, naming the copy `KEYWORD_COPY`. Useful when adding several similar keywords — duplicate one, rename it, and adjust only what's different.

Click any keyword in the list to edit its details in the panel on the right.

KEYWORD DETAIL

Each keyword has the following fields:

FIELD	DESCRIPTION
Keyword	The keyword text as it appears in BASIC source code — e.g. `BORDER`, `MOB`, `ASSEM`.
Token	The token byte value for this keyword in the dialect's token table.
Type	The keyword type — `command`, `function`, `procedure`, `loop`, `conditional`, or `system`.

FIELD	DESCRIPTION
Category	A grouping category for organizational purposes — e.g. `screen`, `graphics`, `sound`, `memory`, `flow`.
Syntax	The full syntax of the keyword, including parameter names — e.g. `BORDER color`.
Desc	A description of what the keyword does. This text appears in hover tooltips in the editor.
Example	An example of the keyword in use.
Notes	Optional additional notes or caveats.

JSON PREVIEW

The **JSON PREVIEW** section at the bottom of the Plugin Editor shows a live, scrollable preview of the plugin's complete JSON representation as you edit. This is the actual content of the `.c64basic` file as it will be saved to disk — useful for verifying the structure of your plugin or copying values for use elsewhere.

The Plugin File Format

Plugin files use the `.c64basic` extension and are plain JSON. Here is a minimal example:

```
{
  "name": "My BASIC Extension",
  "version": "1.0",
  "author": "Your Name",
  "description": "A brief description of this BASIC extension",
  "tokenBase": 203,
  "activationSYS": 2368,
  "keywords": [
    {
      "keyword": "SPRITE",
```

```
        "token": 203,
        "type": "command",
        "category": "graphics",
        "syntax": "SPRITE n, x, y, color",
        "description": "Position and color a sprite",
        "example": "SPRITE 0, 160, 100, 1"
      }
    ]
  }
```

The plugin format also supports `compositeKeywords` — keywords that are represented by two consecutive tokens rather than a single token. For example, VisionBASIC's `LONGPEEK` is stored as the tokens for `LONG` followed by `PEEK`. These are defined separately from the main keywords list so the tokenizer and detokenizer handle them correctly.

Sharing Plugins

To share a plugin with other C64 IDE users, simply send them the `.c64basic` file. They can install it via **Tools → BASIC Dialect → Install Plugin…**, or drop it directly into the Plugins folder revealed by **Reveal Plugins Folder** and restart C64 IDE.

VisionBASIC 1.1

VisionBASIC 1.1 is included with C64 IDE as a fully documented example plugin. It extends standard BASIC V2 with commands for graphics, sprites, sound, memory, flow control, and inline 6502 assembly. It serves both as a working dialect you can use directly, and as a reference for building your own plugins.

To activate VisionBASIC, choose **Tools → BASIC Dialect → VisionBASIC**, then start your program with `SYS 2368` to initialize the extension before using any VisionBASIC keywords.

Appendix A: Keyboard Shortcuts

A complete reference of all keyboard shortcuts in C64 IDE.

File

SHORTCUT	ACTION
⌘N	New file (opens submenu)
⌘O	Open file
⌘S	Save
⌘⇧S	Save As

Edit

SHORTCUT	ACTION
⌘Z	Undo (up to 10 levels, all editors)
⌘⇧Z	Redo
⌘X	Cut
⌘C	Copy
⌘V	Paste
⌘A	Select All

Build

SHORTCUT	ACTION
⌘**R**	Build and Run
⌘⇧**R**	Build and Debug
⌘**B**	Build Only
⌘⌥**D**	Build and Save to D64
⌘⇧**G**	Compile BASIC to ASM
⌘⌃**R**	Run on U64
⌘⌃**L**	Load on U64
⌘**.**	Stop (terminate VICE)

View

SHORTCUT	ACTION
⌘⌥**R**	Toggle Reference Panel
⌘⇧**Y**	Toggle Build Console

Tools

SHORTCUT	ACTION
⌘⇧**E**	Sprite Editor
⌘⇧**D**	Character Set Editor
⌘⌥**G**	Hi-Res Graphics Editor

SHORTCUT	ACTION
⌘⌥**M**	Game Map Editor
⌘⇧**M**	SID Editor
⌘⌥**I**	Image Converter
⌘⇧**K**	D64 Disk Browser
⌘⇧**I**	6502 Disassembler
⌘⇧**P**	PETSCII Character Map
⌘⇧**U**	Number Converter
⌘**,**	Settings

VICE Debugger

SHORTCUT	ACTION
⌘⇧**R**	Build and Debug (launches and connects automatically)

Key to Symbols

SYMBOL	KEY
⌘	Command
⇧	Shift
⌥	Option (Alt)
^	Control
↩	Return

SYMBOL	KEY
## Appendix B: Commodore BASIC V2 Keyword Reference	

A complete reference for all keywords in Commodore BASIC V2, the built-in BASIC interpreter of the Commodore 64. Keywords are listed alphabetically with their syntax and a brief description.

Statements and Commands

KEYWORD	SYNTAX	DESCRIPTION
CLOSE	`CLOSE file#`	Closes a logical file previously opened with OPEN.
CLR	`CLR`	Clears all variables, arrays, and string space. Does not affect the program itself.
CMD	`CMD file#`	Redirects output to a logical file instead of the screen.
CONT	`CONT`	Continues execution of a program that was stopped with STOP or interrupted.
DATA	`DATA value, value, ...`	Defines a list of constants to be read with READ.
DEF FN	`DEF FN name(var) = expression`	Defines a user-defined function.
DIM	`DIM array(size)`	Declares an array and allocates space for it.
END	`END`	Terminates program execution.
FOR	`FOR var = start TO end [STEP n]`	Begins a counted loop. STEP defaults to 1 if omitted.
GET	`GET var$`	Reads a single character from the keyboard buffer without waiting.

KEYWORD	SYNTAX	DESCRIPTION
GET#	`GET# file#, var$`	Reads a single character from a logical file.
GOSUB	`GOSUB line`	Calls a subroutine at the specified line number.
GOTO	`GOTO line`	Jumps to the specified line number unconditionally.
IF	`IF condition THEN statement`	Executes a statement conditionally.
INPUT	`INPUT [prompt;] var`	Reads input from the keyboard, optionally displaying a prompt.
INPUT#	`INPUT# file#, var`	Reads input from a logical file.
LET	`LET var = expression`	Assigns a value to a variable. The LET keyword is optional.
LIST	`LIST [line[-line]]`	Lists the program or a range of lines to the screen.
LOAD	`LOAD "name"[,device[,address]]`	Loads a program from disk or tape.
NEW	`NEW`	Deletes the current program and clears all variables.
NEXT	`NEXT [var]`	Marks the end of a FOR loop and increments the loop variable.
ON	`ON var GOTO/GOSUB line, line, ...`	Branches to one of several lines based on the value of a variable.
OPEN	`OPEN file#, device[, channel[, "name"]]`	Opens a logical file for I/O.
POKE	`POKE address, value`	Writes a byte value to a memory address.
PRINT	`PRINT [expression][, expression]...`	Outputs text and values to the screen.
PRINT#	`PRINT# file#, expression`	

KEYWORD	SYNTAX	DESCRIPTION
		Outputs text and values to a logical file.
READ	`READ var [, var]...`	Reads the next value from a DATA statement into a variable.
REM	`REM comment`	A comment. Everything after REM on the line is ignored.
RESTORE	`RESTORE`	Resets the DATA pointer to the beginning of the first DATA statement.
RETURN	`RETURN`	Returns from a subroutine called with GOSUB.
RUN	`RUN [line]`	Executes the program, optionally starting at a specified line.
SAVE	`SAVE "name"[,device[,type]]`	Saves the current program to disk or tape.
STOP	`STOP`	Halts program execution and prints a BREAK message.
SYS	`SYS address`	Calls a machine language routine at the specified address.
VERIFY	`VERIFY "name"[,device]`	Verifies a saved program against the one in memory.
WAIT	`WAIT address, mask[, xor]`	Pauses execution until a memory location matches a bit pattern.

Numeric Functions

KEYWORD	SYNTAX	DESCRIPTION
ABS	`ABS(x)`	Returns the absolute value of x.

KEYWORD	SYNTAX	DESCRIPTION
ATN	`ATN(x)`	Returns the arctangent of x in radians.
COS	`COS(x)`	Returns the cosine of x (x in radians).
EXP	`EXP(x)`	Returns e raised to the power x.
FN	`FN name(x)`	Calls a user-defined function defined with DEF FN.
FRE	`FRE(x)`	Returns the number of bytes of free memory. The argument is ignored.
INT	`INT(x)`	Returns the largest integer less than or equal to x (floor).
LOG	`LOG(x)`	Returns the natural logarithm of x.
PEEK	`PEEK(address)`	Reads and returns the byte value at a memory address.
POS	`POS(x)`	Returns the current cursor column position. The argument is ignored.
RND	`RND(x)`	Returns a random number between 0 and 1. A negative argument reseeds the generator.
SGN	`SGN(x)`	Returns the sign of x: -1, 0, or 1.
SIN	`SIN(x)`	Returns the sine of x (x in radians).
SQR	`SQR(x)`	Returns the square root of x.
TAN	`TAN(x)`	Returns the tangent of x (x in radians).
USR	`USR(x)`	Calls a user-defined machine language function via the USR vector.

String Functions

KEYWORD	SYNTAX	DESCRIPTION
ASC	`ASC(a$)`	Returns the PETSCII code of the first character of a$.
CHR$	`CHR$(n)`	Returns a one-character string with PETSCII code n.
LEFT$	`LEFT$(a$, n)`	Returns the leftmost n characters of a$.
LEN	`LEN(a$)`	Returns the length of a$ in characters.
MID$	`MID$(a$, start[, len])`	Returns a substring of a$ starting at position start.
RIGHT$	`RIGHT$(a$, n)`	Returns the rightmost n characters of a$.
STR$	`STR$(x)`	Converts a number to its string representation.
TAB	`TAB(n)`	Moves the cursor to column n in a PRINT statement.
VAL	`VAL(a$)`	Converts a string to a numeric value.

Operators

OPERATOR	DESCRIPTION
+	Addition (numeric) or string concatenation.
-	Subtraction or negation.
*	Multiplication.
/	Division.
^	Exponentiation (raise to a power).

OPERATOR	DESCRIPTION
=	Equal to (comparison or assignment).
<	Less than.
>	Greater than.
<=	Less than or equal to.
>=	Greater than or equal to.
<>	Not equal to.
AND	Logical AND (also bitwise AND on integers).
OR	Logical OR (also bitwise OR on integers).
NOT	Logical NOT (also bitwise NOT on integers).

Notes

- Line numbers must be in the range 0–63999.
- Variable names are significant to two characters — `COUNTER` and `COST` are the same variable (`CO`).
- String variables end with `$`, integer variables end with `%`. All other numeric variables are floating point.
- Arrays must be dimensioned with `DIM` before use if they have more than 11 elements (indices 0–10).
- Logical operators also perform bitwise operations on integer values, which is commonly used in C64 programming to manipulate hardware register bits.

Appendix C: 6502 Assembly Instruction Reference

A complete reference for all official 6502 instructions supported by the ca65 assembler. The C64's 6510 processor is fully compatible with the 6502 instruction set.

Addressing Modes

The 6502 supports several addressing modes that determine how an instruction accesses its operand:

MODE	SYNTAX	DESCRIPTION
Implied	`RTS`	No operand — the instruction operates on a fixed register or has no operand.
Accumulator	`ASL A`	Operates directly on the accumulator.
Immediate	`LDA #$42`	Operand is a literal byte value.
Zero Page	`LDA $42`	Operand is an address in the zero page ($0000–$00FF). Faster and smaller than absolute.
Zero Page, X	`LDA $42,X`	Zero page address plus X register offset.
Zero Page, Y	`LDA $42,Y`	Zero page address plus Y register offset.
Absolute	`LDA $1234`	Operand is a full 16-bit address.
Absolute, X	`LDA $1234,X`	Absolute address plus X register offset.
Absolute, Y	`LDA $1234,Y`	Absolute address plus Y register offset.
Indirect	`JMP ($1234)`	Jumps to the address stored at the given address.

MODE	SYNTAX	DESCRIPTION
Indirect, X	`LDA ($42,X)`	Zero page address plus X, then indirect.
Indirect, Y	`LDA ($42),Y`	Indirect through zero page address, then plus Y.
Relative	`BEQ label`	Signed 8-bit offset from the next instruction. Used by branch instructions only.

Load and Store

MNEMONIC	FULL NAME	DESCRIPTION
LDA	Load Accumulator	Loads a byte into the accumulator. Sets N and Z flags.
LDX	Load X Register	Loads a byte into the X register. Sets N and Z flags.
LDY	Load Y Register	Loads a byte into the Y register. Sets N and Z flags.
STA	Store Accumulator	Stores the accumulator to a memory address.
STX	Store X Register	Stores the X register to a memory address.
STY	Store Y Register	Stores the Y register to a memory address.

Transfer

MNEMONIC	FULL NAME	DESCRIPTION
TAX	Transfer A to X	Copies the accumulator into X. Sets N and Z flags.

MNEMONIC	FULL NAME	DESCRIPTION
TAY	Transfer A to Y	Copies the accumulator into Y. Sets N and Z flags.
TXA	Transfer X to A	Copies X into the accumulator. Sets N and Z flags.
TYA	Transfer Y to A	Copies Y into the accumulator. Sets N and Z flags.
TSX	Transfer SP to X	Copies the stack pointer into X. Sets N and Z flags.
TXS	Transfer X to SP	Copies X into the stack pointer. Does not affect flags.

Stack

MNEMONIC	FULL NAME	DESCRIPTION
PHA	Push Accumulator	Pushes the accumulator onto the stack.
PLA	Pull Accumulator	Pulls a byte from the stack into the accumulator. Sets N and Z flags.
PHP	Push Processor Status	Pushes the processor flags register onto the stack.
PLP	Pull Processor Status	Pulls a byte from the stack into the flags register.

Arithmetic

MNEMONIC	FULL NAME	DESCRIPTION
ADC	Add with Carry	Adds a byte and the carry flag to the accumulator. Sets N, V, Z, C flags.
SBC		

MNEMONIC	FULL NAME	DESCRIPTION
	Subtract with Carry	Subtracts a byte and the inverse of carry from the accumulator. Sets N, V, Z, C flags.
INC	Increment Memory	Increments a byte in memory by 1. Sets N and Z flags.
INX	Increment X	Increments X by 1. Sets N and Z flags.
INY	Increment Y	Increments Y by 1. Sets N and Z flags.
DEC	Decrement Memory	Decrements a byte in memory by 1. Sets N and Z flags.
DEX	Decrement X	Decrements X by 1. Sets N and Z flags.
DEY	Decrement Y	Decrements Y by 1. Sets N and Z flags.

Logical

MNEMONIC	FULL NAME	DESCRIPTION
AND	Logical AND	Bitwise AND of the accumulator with a byte. Sets N and Z flags.
ORA	Logical OR	Bitwise OR of the accumulator with a byte. Sets N and Z flags.
EOR	Exclusive OR	Bitwise XOR of the accumulator with a byte. Sets N and Z flags.
BIT	Bit Test	ANDs the accumulator with a memory byte to set flags without storing the result. Sets N, V, Z flags.

Shift and Rotate

MNEMONIC	FULL NAME	DESCRIPTION
ASL	Arithmetic Shift Left	Shifts left by one bit, shifting 0 into bit 0 and bit 7 into carry. Sets N, Z, C flags.
LSR	Logical Shift Right	Shifts right by one bit, shifting 0 into bit 7 and bit 0 into carry. Sets N, Z, C flags.
ROL	Rotate Left	Shifts left by one bit through carry — carry goes into bit 0, bit 7 goes into carry. Sets N, Z, C flags.
ROR	Rotate Right	Shifts right by one bit through carry — carry goes into bit 7, bit 0 goes into carry. Sets N, Z, C flags.

Compare

MNEMONIC	FULL NAME	DESCRIPTION
CMP	Compare Accumulator	Compares the accumulator with a byte by subtracting without storing. Sets N, Z, C flags.
CPX	Compare X Register	Compares X with a byte. Sets N, Z, C flags.
CPY	Compare Y Register	Compares Y with a byte. Sets N, Z, C flags.

Branch

Branch instructions jump to a nearby address (within −128 to +127 bytes of the next instruction) if their condition is met.

MNEMONIC	FULL NAME	BRANCHES IF...
BCC	Branch if Carry Clear	Carry flag = 0
BCS	Branch if Carry Set	Carry flag = 1
BEQ	Branch if Equal	Zero flag = 1
BMI	Branch if Minus	Negative flag = 1
BNE	Branch if Not Equal	Zero flag = 0
BPL	Branch if Plus	Negative flag = 0
BVC	Branch if Overflow Clear	Overflow flag = 0
BVS	Branch if Overflow Set	Overflow flag = 1

Jump and Subroutine

MNEMONIC	FULL NAME	DESCRIPTION
JMP	Jump	Jumps to the specified address unconditionally. Supports absolute and indirect modes.
JSR	Jump to Subroutine	Pushes the return address onto the stack and jumps to the specified address.
RTS	Return from Subroutine	Pulls the return address from the stack and jumps to it (plus one).
RTI	Return from Interrupt	Pulls the flags and return address from the stack. Used at the end of interrupt handlers.
BRK	Break	Triggers a software interrupt. Pushes the PC and flags onto the stack and jumps via the BRK vector.

Flag Control

MNEMONIC	FULL NAME	DESCRIPTION
CLC	Clear Carry	Sets the carry flag to 0.
SEC	Set Carry	Sets the carry flag to 1.
CLD	Clear Decimal	Sets the decimal mode flag to 0. (Decimal mode has no effect on the 6510.)
SED	Set Decimal	Sets the decimal mode flag to 1.
CLI	Clear Interrupt Disable	Enables maskable interrupts (IRQ).
SEI	Set Interrupt Disable	Disables maskable interrupts (IRQ).
CLV	Clear Overflow	Sets the overflow flag to 0.

Miscellaneous

MNEMONIC	FULL NAME	DESCRIPTION
NOP	No Operation	Does nothing for 2 cycles. Used for timing delays or as padding.

Processor Flags

FLAG	BIT	DESCRIPTION
N	7	Negative — set if the result of an operation has bit 7 set.
V	6	Overflow — set if a signed arithmetic operation overflowed.

FLAG	BIT	DESCRIPTION
B	4	Break — set when BRK is executed.
D	3	Decimal — enables BCD arithmetic mode (non-functional on the 6510).
I	2	Interrupt Disable — when set, maskable interrupts are ignored.
Z	1	Zero — set if the result of an operation is zero.
C	0	Carry — set by arithmetic operations, shifts, and rotates.

Notes

- Always clear the carry flag with `CLC` before `ADC`, and set it with `SEC` before `SBC`, unless you specifically want to include the carry from a previous operation.
- The zero page ($0000–$00FF) is valuable memory on the C64 — zero page instructions are faster (one fewer cycle) and one byte smaller than absolute addressing. Use it for frequently accessed variables.
- `JSR` pushes the address of the last byte of the JSR instruction onto the stack. `RTS` pulls this address and adds one, returning to the instruction after the JSR.
- On the C64, `SEI` is used to disable interrupts during time-critical code sections, and `CLI` to re-enable them. Always re-enable interrupts with `CLI` when finished.

Appendix D: Common KERNAL Routines

The C64's KERNAL ROM provides a set of well-documented subroutines for common tasks — character I/O, file operations, screen control, and system utilities. Call them from assembly using `JSR address`. The routines listed here are the ones most commonly used in C64 programming.

Input and output registers are described using standard 6502 register notation: **A** (accumulator), **X**, **Y**, **C** (carry flag).

Character I/O

CHROUT — $FFD2

Output a character to the current output device (default: screen).

Input	A = PETSCII character code to output
Output	C set on error
Preserves	A

The default output device is the screen. After `OPEN` and `CMD`, output is redirected to a logical file. Use `PRINT#` equivalent logic to send formatted output to files.

```
LDA #65         ; PETSCII 'A'
JSR CHROUT      ; print it
```

CHRIN — $FFCF

Input a character from the current input device (default: keyboard).

Input	None
Output	A = PETSCII character code received
Preserves	X, Y

Waits for a character to be available before returning.

GETIN — $FFE4

Read a character from the keyboard buffer without waiting.

Input	None
Output	A = character from buffer, or 0 if buffer empty
Preserves	X, Y

Unlike CHRIN, GETIN returns immediately. Check for A = 0 to detect an empty buffer. Equivalent to BASIC's `GET`.

```
JSR GETIN       ; get a key
CMP #0          ; was a key pressed?
BEQ no_key      ; no — buffer was empty
```

CLRCHN — $FFCC

Restore default I/O channels (keyboard input, screen output).

Input	None
Output	None

Call after using CMD or CHKOUT to redirect I/O back to the default devices.

Screen Control

CLRSCR — $E544

Clear the screen and move the cursor to the home position.

Input	None
Output	None

> **Note:** This is a ROM routine address, not a KERNAL jump table entry. It may vary between ROM versions. The portable alternative is to output `CHR$(147)` via CHROUT.

```
JSR $E544       ; clear screen
```

PLOT — $FFF0

Read or set the cursor position.

Input	C = 0 to set position, C = 1 to read position
Input (set)	X = column (0–39), Y = row (0–24)
Output (read)	X = current column, Y = current row

```
; Set cursor to column 10, row 5
LDX #10
LDY #5
CLC
JSR PLOT
```

File I/O

SETLFS — $FFBA

Set up a logical file for I/O.

Input	A = logical file number, X = device number, Y = secondary address
Output	None

Call before OPEN. Common device numbers: 0 = keyboard, 1 = tape, 3 = screen, 4–7 = printer, 8–11 = disk drive.

```
LDA #1          ; logical file 1
LDX #8          ; device 8 (disk)
LDY #0          ; secondary address 0
JSR SETLFS
```

SETNAM — $FFBD

Set the filename for the next OPEN or LOAD operation.

Input	A = filename length, X = low byte of filename address, Y = high byte of filename address
Output	None

```
LDA #8          ; filename length
LDX #<filename  ; low byte of address
LDY #>filename  ; high byte of address
JSR SETNAM
```

```
filename: .byte "MYFILE.PRG"
```

OPEN — $FFC0

Open a logical file.

Input	None (uses values set by SETLFS and SETNAM)
Output	C set on error, A = error code

Call SETLFS and SETNAM before OPEN. After opening, use CHKOUT or CHKIN to direct I/O to the file.

CLOSE — $FFC3

Close a logical file.

Input	A = logical file number to close
Output	C set on error

Always close files when finished to free KERNAL resources.

```
LDA #1          ; close logical file 1
JSR CLOSE
```

CHKOUT — $FFC9

Set a logical file as the current output channel.

Input	X = logical file number
Output	C set on error

After CHKOUT, all CHROUT calls go to this file. Call CLRCHN to restore screen output.

CHKIN — $FFC6

Set a logical file as the current input channel.

Input	X = logical file number
Output	C set on error

After CHKIN, all CHRIN calls read from this file. Call CLRCHN to restore keyboard input.

LOAD — $FFD5

Load a file from a device into memory.

Input	A = 0 (load) or 1 (verify), X = low byte of load address, Y = high byte of load address
Output	C set on error, X/Y = address of last byte loaded + 1

Uses the filename and device set by SETNAM and SETLFS. If the file has a two-byte load address header (PRG format), the header address is used instead of the X/Y values when the secondary address is 1.

SAVE — $FFD8

Save memory to a device.

Input	A = zero page address of start address pointer, X = low byte of end address + 1, Y = high byte of end address + 1
Output	C set on error

System

RESTOR — $FF8A

Restore the default KERNAL vectors.

Input	None
Output	None

Resets all KERNAL indirect vectors to their default values. Useful for recovering from a corrupted system state.

MEMTOP — $FF99

Read or set the top of available memory.

Input	C = 0 to set, C = 1 to read
Input (set)	X = low byte, Y = high byte
Output (read)	X = low byte, Y = high byte

MEMBOT — $FF9C

Read or set the bottom of available memory.

Input	C = 0 to set, C = 1 to read
Input (set)	X = low byte, Y = high byte
Output (read)	X = low byte, Y = high byte

SCNKEY — $FF9F

Scan the keyboard and update the keyboard buffer.

Input	None
Output	None

Normally called by the interrupt handler. Call manually if interrupts are disabled and you need keyboard input.

SETTIM — $FFDB

Set the software clock (jiffy clock).

Input	A = high byte, X = middle byte, Y = low byte
Output	None

The jiffy clock increments 60 times per second (NTSC) or 50 times per second (PAL).

RDTIM — $FFDE

Read the software clock (jiffy clock).

Input	None
Output	A = high byte, X = middle byte, Y = low byte

```
JSR RDTIM       ; read the clock
STY timer_lo    ; save low byte
STX timer_mid   ; save middle byte
STA timer_hi    ; save high byte
```

Quick Reference Table

ROUTINE	ADDRESS	DESCRIPTION
CHKIN	$FFC6	Set input channel
CHKOUT	$FFC9	Set output channel
CHRIN	$FFCF	Input character
CHROUT	$FFD2	Output character
CLRCHN	$FFCC	Restore default channels
CLOSE	$FFC3	Close logical file
GETIN	$FFE4	Get character from keyboard buffer
LOAD	$FFD5	Load file from device
MEMBOT	$FF9C	Read/set bottom of memory
MEMTOP	$FF99	Read/set top of memory

ROUTINE	ADDRESS	DESCRIPTION
OPEN	$FFC0	Open logical file
PLOT	$FFF0	Read/set cursor position
RDTIM	$FFDE	Read jiffy clock
RESTOR	$FF8A	Restore KERNAL vectors
SAVE	$FFD8	Save memory to device
SCNKEY	$FF9F	Scan keyboard
SETLFS	$FFBA	Set logical file parameters
SETNAM	$FFBD	Set filename
SETTIM	$FFDB	Set jiffy clock
## Appendix E: C64 Color Reference		

The Commodore 64 supports 16 colors, numbered 0–15. These color values are used throughout C64 programming — in POKE statements, VIC-II register writes, sprite color settings, and more.

Color Table

#	NAME	BASIC POKE	VIC-II REGISTER VALUE
0	Black	`POKE 53281,0`	$00
1	White	`POKE 53281,1`	$01
2	Red	`POKE 53281,2`	$02
3	Cyan	`POKE 53281,3`	$03
4	Purple	`POKE 53281,4`	$04

#	NAME	BASIC POKE	VIC-II REGISTER VALUE
5	Green	`POKE 53281,5`	$05
6	Blue	`POKE 53281,6`	$06
7	Yellow	`POKE 53281,7`	$07
8	Orange	`POKE 53281,8`	$08
9	Brown	`POKE 53281,9`	$09
10	Light Red	`POKE 53281,10`	$0A
11	Dark Grey	`POKE 53281,11`	$0B
12	Medium Grey	`POKE 53281,12`	$0C
13	Light Green	`POKE 53281,13`	$0D
14	Light Blue	`POKE 53281,14`	$0E
15	Light Grey	`POKE 53281,15`	$0F

Common Color Registers

ADDRESS	DEC	DESCRIPTION
$D020	53280	Border color
$D021	53281	Background color 0 (screen background)
$D022	53282	Background color 1 (multicolor mode)
$D023	53283	Background color 2 (multicolor mode)
$D024	53284	Background color 3 (multicolor mode)
$D800–$DBE7	55296–56295	Color RAM — one byte per screen cell (40×25)

Setting Colors in BASIC

```
POKE 53280, 0   : REM BLACK BORDER
POKE 53281, 6   : REM BLUE BACKGROUND
POKE 646, 1     : REM WHITE TEXT COLOR
```

Setting Colors in Assembly

```
LDA #0          ; black
STA $D020       ; set border color

LDA #6          ; blue
STA $D021       ; set background color
```

Appendix F: Common C64 Memory Locations

A quick reference for the most frequently used memory addresses in C64 programming. Addresses are given in both hexadecimal and decimal.

System Areas

ADDRESS	DEC	DESCRIPTION
$0000–$00FF	0–255	Zero page — fast access memory, used heavily by KERNAL and BASIC
$0100–$01FF	256–511	6510 hardware stack
$0200–$02FF	512–767	KERNAL and BASIC input buffer and workspace
	768–831	KERNAL indirect vectors (editable)

ADDRESS	DEC	DESCRIPTION
$0300–$033F		
$033C–$03FB	828–1019	Cassette buffer (safe to use as scratch space if not using tape)
$0400–$07E7	1024–2023	Screen RAM — 40×25 character codes
$0800–$9FFF	2048–40959	BASIC program area (38911 bytes free on a stock C64)

BASIC System Variables

ADDRESS	DEC	DESCRIPTION
$0028/$0029	40/41	Start of BASIC program (low/high byte)
$002B/$002C	43/44	Start of variable storage (low/high byte)
$002D/$002E	45/46	Start of array storage (low/high byte)
$0030/$0031	48/49	End of array storage / start of string storage (low/high byte)
$0033/$0034	51/52	Start of string storage (low/high byte)
$0037/$0038	55/56	Top of memory available to BASIC (low/high byte)
$0090	144	STATUS — I/O status byte
$0091	145	STOP key flag (127 = STOP pressed)
$009D	157	Output flag (0 = screen, 128 = direct mode)
$00C5	197	Current key pressed (64 = no key)
$0286	646	Current text color
$0287	647	Color under cursor

ADDRESS	DEC	DESCRIPTION
$D3	211	Current cursor column
$D6	214	Current cursor row

VIC-II Registers ($D000–$D3FF)

ADDRESS	DEC	DESCRIPTION
$D000/$D001	53248/53249	Sprite 0 X/Y position
$D002/$D003	53250/53251	Sprite 1 X/Y position
$D004/$D005	53252/53253	Sprite 2 X/Y position
$D006/$D007	53254/53255	Sprite 3 X/Y position
$D008/$D009	53256/53257	Sprite 4 X/Y position
$D00A/$D00B	53258/53259	Sprite 5 X/Y position
$D00C/ $D00D	53260/53261	Sprite 6 X/Y position
$D00E/$D00F	53262/53263	Sprite 7 X/Y position
$D010	53264	Sprites MSB of X position (bit 8 of X for each sprite)
$D011	53265	VIC-II control register 1 (screen on, bitmap mode, raster bit 8, etc.)
$D012	53266	Current raster line / raster interrupt line
$D015	53269	Sprite enable register (bit per sprite)
$D016	53270	VIC-II control register 2 (multicolor mode, column count)
$D017	53271	Sprite vertical expansion (bit per sprite)
$D018	53272	

ADDRESS	DEC	DESCRIPTION
		Memory control register (screen and character set base addresses)
$D019	53273	Interrupt request register
$D01A	53274	Interrupt enable register
$D01B	53275	Sprite display priority (0=sprite in front, 1=sprite behind background)
$D01C	53276	Sprite multicolor enable (bit per sprite)
$D01D	53277	Sprite horizontal expansion (bit per sprite)
$D01E	53278	Sprite-to-sprite collision register
$D01F	53279	Sprite-to-background collision register
$D020	53280	Border color
$D021	53281	Background color 0
$D022	53282	Background color 1 (multicolor)
$D023	53283	Background color 2 (multicolor)
$D024	53284	Background color 3 (multicolor)
$D025	53285	Sprite multicolor 0 (shared)
$D026	53286	Sprite multicolor 1 (shared)
$D027–$D02E	53287–53294	Sprite 0–7 individual colors
$D800–$DBE7	55296–56295	Color RAM (one nybble per screen cell)

SID Registers ($D400–$D7FF)

ADDRESS	DEC	DESCRIPTION
$D400/$D401	54272/54273	Voice 1 frequency (low/high byte)
$D402	54274	Voice 1 pulse width low byte
$D403	54275	Voice 1 pulse width high nybble
$D404	54276	Voice 1 control register (waveform, gate, etc.)
$D405	54277	Voice 1 attack/decay
$D406	54278	Voice 1 sustain/release
$D407/$D408	54279/54280	Voice 2 frequency (low/high byte)
$D409	54281	Voice 2 pulse width low byte
$D40A	54282	Voice 2 pulse width high nybble
$D40B	54283	Voice 2 control register
$D40C	54284	Voice 2 attack/decay
$D40D	54285	Voice 2 sustain/release
$D40E/$D40F	54286/54287	Voice 3 frequency (low/high byte)
$D410	54288	Voice 3 pulse width low byte
$D411	54289	Voice 3 pulse width high nybble
$D412	54290	Voice 3 control register
$D413	54291	Voice 3 attack/decay
$D414	54292	Voice 3 sustain/release
$D415/$D416	54293/54294	Filter cutoff frequency (low 3 bits / high 8 bits)
$D417	54295	Filter resonance and voice routing
$D418	54296	Volume and filter mode

ADDRESS	DEC	DESCRIPTION
$D41B	54299	Voice 3 oscillator output (read only)
$D41C	54300	Voice 3 envelope output (read only)

CIA Registers

CIA 1 ($DC00–$DCFF) — KEYBOARD, JOYSTICK, JIFFY CLOCK

ADDRESS	DEC	DESCRIPTION
$DC00	56320	Port A — joystick 2, keyboard column select
$DC01	56321	Port B — joystick 1, keyboard row read
$DC04/$DC05	56324/56325	Timer A (low/high byte)
$DC06/$DC07	56326/56327	Timer B (low/high byte)
$DC08	56328	Time of day — tenths of seconds
$DC09	56329	Time of day — seconds
$DC0A	56330	Time of day — minutes
$DC0B	56331	Time of day — hours (bit 7 = AM/PM)
$DC0D	56333	Interrupt control register

CIA 2 ($DD00–$DDFF) — SERIAL BUS, NMI, VIC-II BANK

ADDRESS	DEC	DESCRIPTION
$DD00	56576	Port A — serial bus, VIC-II memory bank select (bits 0–1)
$DD01	56577	Port B — serial bus
$DD0D	56589	NMI interrupt control register

ROM Areas

ADDRESS	DEC	DESCRIPTION
$A000–$BFFF	40960–49151	BASIC ROM
$D000–$DFFF	53248–57343	I/O area (or character ROM when bank-switched)
$E000–$FFFF	57344–65535	KERNAL ROM
$FF48	65352	IRQ handler entry point
$FFFE/$FFFF	65534/65535	IRQ/BRK vector
$FFFA/$FFFB	65530/65531	NMI vector
$FFFC/$FFFD	65532/65533	RESET vector