```
10 PRINT "HELLO"
20 GOTO 10
RUN
```

# NOVABASIC

## User Guide v1.0

*For the e6502 Virtual Computer*

Derived from EhBASIC 2.22p5 | 2026 Edition

---

**MOS 6502 | 320×200 Graphics | 4-Voice Sound | 512KB Expansion**

**NovaBASIC v1.0 User Guide**

For the e6502 Virtual Computer — 2026 Edition

---



e6502 is an open source project.



---

Typeset with LaTeX and the `lmodern` font family.
Document class: `book`, 11 pt, A4.
Build toolchain: `latexmk` with `pdflatex`.

# Contents

# Foreword

There is something quietly extraordinary about a chip that fits in the palm of your hand and changed the world. The MOS 6502, introduced in 1975, powered the Apple II, the Commodore 64, the Nintendo Entertainment System, the Atari 2600, and the BBC Micro. It put a computer on the desk of a generation that had never seen one before. It was cheap enough for a hobbyist to buy, simple enough to understand completely, and fast enough to make things happen on screen that felt genuinely alive. Whole careers began on machines built around that little chip. So did countless imaginations.

The years moved on. Processors grew faster, wider, and vastly more complex. But the 6502 never really went away — it kept shipping in embedded systems, and it kept living in the memories of the people who learned on it. There is a reason emulators, homebrew cartridges, and new assembler projects for the 6502 keep appearing in 2026: programming it is a complete, intimate experience. You can hold the whole machine in your head at once.

The e6502 virtual computer is not a museum exhibit. It takes the honest architecture of the original — 64 KB of address space, stack at `$0100`–`$01FF`, clean interrupt model — and pairs it with hardware that no real 8-bit machine ever had: a 320×200 pixel bitmap with 16 colors, sixteen hardware sprites each 16×16 pixels and multicolor, a four-voice synthesizer with ADSR envelopes and five waveforms, 512 KB of banked expansion memory, and file I/O. It is the machine you always wished you had back then.

NovaBASIC is the language that lives on it. Built on Lee Davison's *Enhanced BASIC 2.22p5* — itself a carefully crafted piece of work — NovaBASIC keeps everything that made classic BASIC approachable: line numbers, `PRINT`, `FOR` / `NEXT`, `GOSUB`, the immediacy of typing a line and seeing it run. Then it adds the commands to drive all that new hardware, without hiding the machine from you.

Whether you are learning to program for the first time, building a retro game, exploring procedural graphics, or just looking for a creative sandbox that rewards curiosity — this is the right place. Boot it up. Type something. Watch it run.

Welcome.

*February 2026*

Chapter 1

# Welcome to NovaBASIC

Every great journey starts with a single command.
*Unknown programmer, circa 1982*

## 1.1 What NovaBASIC Is

NovaBASIC v1.0 is a modernized 6502 BASIC interpreter for the e6502 virtual computer. It is derived from Lee Davison's *Enhanced BASIC 2.22p5* (EhBASIC), one of the cleanest and most complete open-source 6502 BASIC implementations ever written. NovaBASIC keeps the entire EhBASIC core — every numeric function, every string operation, the full `DO` / `LOOP` and `FOR` / `NEXT` machinery — and adds a hardware command layer that lets you drive the e6502's graphics, sound, file system, and expansion memory directly from BASIC.

The programming model is classic line-numbered BASIC. You type a line with a number at the front, press Return, and it is stored in the program. You type `RUN` and it executes from the lowest line number. Simple, immediate, and still deeply satisfying.

When NovaBASIC boots, the screen clears to a blue background with white text and you see:

```
NovaBASIC v1.0
Derived from EhBASIC 2.22p5
xxxxx BASIC bytes free
```

The number shown is the amount of free BASIC program memory. The cursor waits for your first command.

Boot sets the screen background to color 6 (blue) and the foreground text color to color 1 (white). You can change these at any time with `COLOR fg,bg`.

## 1.2 What You Can Do

Here is what NovaBASIC puts at your fingertips:

- Write and run classic line-numbered BASIC programs in the tradition of the Commodore 64, Apple II, and BBC Micro.
- Draw directly to a **320×200 pixel bitmap** with **16 colors** using `PLOT`, `LINE`, `CIRCLE`, `RECT`, `FILL`, and `PAINT`.

- Animate up to **16 hardware sprites**, each 16×16 pixels and multicolor, with per-sprite shape, position, and flip control.
- Play **4-voice synthesized sound** with ADSR envelopes and five selectable waveforms — square, sawtooth, triangle, sine, and noise.
- Save and load programs and data files with `SAVE`, `LOAD`, `DIR`, and `DEL`.
- Access **512 KB of banked expansion memory** for large data sets, tile maps, sprite sheets, or music sequences.
- Drop into **6502 assembly language** via `CALL` and `POKE` / `PEEK` for performance-critical inner loops.

## 1.3 How to Read This Guide

This manual is organized so each chapter builds on the previous one. You do not need to read it straight through — jump to whichever section you need — but if you are new to NovaBASIC the order makes sense:

**Chapters 1–3** Getting started, the edit-run workflow, and the core language: variables, arrays, control flow, operators, and built-in functions.

**Chapters 4–5** Graphics and sprites; sound and music. These two chapters cover everything you need to build a real game or demo.

**Chapters 6–7** Expansion memory and low-level access. Read these when your projects outgrow the built-in 64 KB address space or when you want to call hand-written assembly routines.

**Appendices** Complete command reference, memory map, hardware register summary, error codes, and system limits.

Cross-references appear as "see Section X.Y" or "see Appendix A." Every command name is typeset in `this style` throughout the text.

## 1.4 Your First Program

Let us get something on screen right now. Type each line exactly as shown, pressing Return after each one, then type `RUN` and press Return.

```
10 PRINT "HELLO, NOVABASIC!"
20 FOR I=1 TO 5
30   PRINT "COUNT: ";I
40 NEXT I
RUN
```

**Expected output:**

```
HELLO, NOVABASIC!
COUNT: 1
COUNT: 2
COUNT: 3
COUNT: 4
COUNT: 5
```

Six lines of output, then the cursor returns to the `Ready` prompt. Congratulations — you have just run your first NovaBASIC program.

Chapter 2

# Your First Session

> Tell me and I forget. Teach me and I remember. Involve me and I learn.
>
> *Benjamin Franklin*

The fastest way to get comfortable with NovaBASIC is to use it. This chapter walks you through everything you need for a productive first session: entering programs, editing them, saving and loading files, and building the habits that will serve you on every project after this one.

## 2.1 The Edit-Run Cycle

NovaBASIC stores your program as an ordered list of numbered lines. When you type a line beginning with a number and press Return, that line is added to the program in the correct position. When you type `RUN`, execution begins at the lowest line number and proceeds in order.

This is the fundamental loop:

**TYPE A LINE ⟶ LIST ⟶ RUN ⟶ FIX ⟶ RUN**

Here is a small program to enter right now:

```
10 PRINT "NOVABASIC IS READY"
20 X = 7
30 PRINT "SEVEN SQUARED IS "; X*X
```

Type `LIST` to see the program back. Type `RUN` to execute it. Now change line 20 by retyping it with a new value:

```
20 X = 12
```

The old line 20 is replaced. Type `RUN` again and the result changes. To delete a line entirely, type its number alone and press Return:

```
30
```

Line 30 is gone. `LIST` confirms it.

## 2.2 Essential Editing Commands

| Command | What It Does |
|---|---|
| `LIST` | Display the entire program currently in memory. |
| `LIST 20-40` | Display only lines 20 through 40 (inclusive). |
| `NEW` | Clear the program from memory. Cannot be undone. |
| `RUN` | Execute the program starting from the lowest line number. |
| `CONT` | Continue execution after a `STOP` statement. |

**Warning**

`NEW` erases everything in memory immediately. Save your work with `SAVE` before typing `NEW` if you want to keep it.

## 2.3 Saving and Loading

Programs are saved to and loaded from the virtual file system using four commands:

`SAVE "name"` Write the current program to disk under the given name. The `.bas` extension is added automatically.

`LOAD "name"` Load a saved program from disk into memory, replacing anything currently there.

`DIR` List all saved BASIC programs.

`DEL "name"` Delete a saved program. This cannot be undone.

Filenames may contain letters (`A-Z`, `a-z`), digits (`0-9`), underscores, hyphens, and dots. Maximum length is 63 characters. Names are case-sensitive on disk.

Here is a complete save-and-reload sequence:

```
SAVE "MYPROG"
DIR
NEW
LOAD "MYPROG"
RUN
```

`DIR` shows `MYPROG.bas` in the listing. After `LOAD` and `RUN`, the program executes as if you had just typed it in.

## 2.4 Working Efficiently

Good habits established early make everything easier later.

- **Number lines by tens.** Use line numbers 10, 20, 30... rather than 1, 2, 3. This leaves room to insert new lines between existing ones without renumbering.
- **Use `REM` freely while learning.** A comment line costs a little memory but saves a lot of confusion. You can remove them once the code is stable.
- **Build and run in small steps.** Add a few lines, `RUN`, verify the output, add

more. Chasing bugs through a hundred lines you typed without testing is no fun.

- **Group lines by function.** A common convention: 100s for initialisation, 200s for input, 300s for game logic, 800s for output routines, 900s for cleanup and quit. Any scheme that makes sense to you is fine; the important thing is to pick one and use it.

## 2.5 The Round-Trip Exercise

The following exercise takes you through writing, saving, clearing, reloading, and running a program. Every step matters — do not skip any of them.

```
10 PRINT "ROUND TRIP COMPLETE"
```

Then, at the prompt (no line number):

```
SAVE "ROUNDTRIP"
NEW
DIR
LOAD "ROUNDTRIP"
RUN
```

**Expected:**

- After `SAVE`: the prompt returns immediately with no error.
- After `NEW`: `LIST` shows nothing.
- After `DIR`: `ROUNDTRIP.bas` appears in the listing.
- After `LOAD` and `RUN`: the screen prints `ROUND TRIP COMPLETE`.

If you see `ROUND TRIP COMPLETE` at the end, you have mastered the complete NovaBASIC workflow. Everything else is built on top of exactly this.

Chapter 3

# Language Fundamentals

> Simplicity is the ultimate sophistication.
>
> *Leonardo da Vinci*

NovaBASIC inherits a rich language from EhBASIC 2.22p5 and extends it with hardware-specific commands. This chapter covers the core language features you will use in almost every program: variables, arrays, control flow, operators, built-in functions, and direct memory access.

## 3.1 Variables

NovaBASIC has two kinds of variable: **numeric** and **string**.

**Numeric variables** hold floating-point numbers. Names begin with a letter and may contain letters and digits (e.g. `A`, `SCORE`, `X1`, `HITCOUNT`).

**String variables** hold text. Names end with a dollar sign (e.g. `N$`, `NAME$`, `MSG$`).

Variable names are **case-insensitive**: `SCORE` and `score` refer to the same variable. Numeric variables default to `0`; string variables default to the empty string. You do not need to declare a variable before using it.

```
10 SCORE = 0
20 NAME$ = "PLAYER ONE"
30 LIVES = 3
40 PRINT NAME$; " HAS "; LIVES; " LIVES"
50 SCORE = SCORE + 100
60 PRINT "SCORE: "; SCORE
```

## 3.2 Arrays

Use `DIM` to declare an array before using it. The default lower bound is 0, so `DIM A(10)` creates 11 elements: `A(0)` through `A(10)`.

```
10 DIM A(10)
20 FOR I=0 TO 10
30   A(I) = I * I
40 NEXT I
50 FOR I=0 TO 10
60   PRINT "A(";I;") = ";A(I)
70 NEXT I
```

Two-dimensional arrays work the same way:

```
10 DIM GRID(7,7)
20 FOR R=0 TO 7
30   FOR C=0 TO 7
40     GRID(R,C) = R*8 + C
50   NEXT C
60 NEXT R
```

> String arrays are also supported: `DIM LABEL$(9)` creates ten string slots.

## 3.3 Control Flow

### 3.3.1 FOR / NEXT Loops

The workhorse of NovaBASIC iteration. The optional `STEP` clause sets the increment; if omitted, it defaults to 1.

```
10 FOR I = 1 TO 10
20   PRINT I; " ";
30 NEXT I
40 PRINT
50 REM count down with STEP
60 FOR N = 10 TO 1 STEP -1
70   PRINT N; " ";
80 NEXT N
```

### 3.3.2 IF / THEN / ELSE

```
10 INPUT "ENTER A NUMBER: "; N
20 IF N > 100 THEN PRINT "BIG" ELSE PRINT "SMALL"
30 IF N = 42 THEN PRINT "THE ANSWER"
```

`THEN` may be followed by a statement or a line number (as a `GOTO` shorthand). `ELSE` is optional.

### 3.3.3 DO / LOOP

`DO` / `LOOP` supports four variants for flexible looping:

```
10 X = 1
20 DO
30   PRINT X
40   X = X + 1
50 LOOP WHILE X <= 5

100 Y = 10
110 DO UNTIL Y = 0
120   PRINT Y
130   Y = Y - 1
140 LOOP
```

### 3.3.4 GOSUB / RETURN

Use subroutines to avoid repeating code. `GOSUB` branches to a line number and `RETURN` comes back:

```
10 GOSUB 1000
20 GOSUB 1000
30 END
1000 REM draw border subroutine
1010 PRINT "----------"
1020 RETURN
```

### 3.3.5 GOTO

`GOTO` unconditionally jumps to a line number. Use it sparingly; `GOSUB` / `RETURN` and structured loops are usually cleaner.

## 3.4 Operators

| Operator | Type | Description |
|---|---|---|
| `+` | Arithmetic | Addition |
| `-` | Arithmetic | Subtraction or unary negation |
| `*` | Arithmetic | Multiplication |
| `/` | Arithmetic | Division |
| `^` | Arithmetic | Exponentiation (`2^8` = 256) |
| `=` | Comparison | Equal to |
| `<` | Comparison | Less than |
| `>` | Comparison | Greater than |
| `<=` | Comparison | Less than or equal to |
| `>=` | Comparison | Greater than or equal to |
| `<>` | Comparison | Not equal to |
| `AND` | Logical | Logical (bitwise) AND |
| `OR` | Logical | Logical (bitwise) OR |
| `NOT` | Logical | Logical (bitwise) NOT |
| `EOR` | Logical | Exclusive OR |
| `«` | Bit | Left shift (`LSHIFT`) |
| `»` | Bit | Right shift (`RSHIFT`) |
| `BITSET` | Bit | Set a bit in an integer |
| `BITCLR` | Bit | Clear a bit in an integer |
| `BITTST(n,b)` | Bit | Test bit $b$ of integer $n$ |

## 3.5 Built-in Functions

### 3.5.1 Numeric Functions

| Function | Description |
|---|---|
| `INT(x)` | Truncate to integer (towards zero) |
| `ABS(x)` | Absolute value |
| `SGN(x)` | Sign: $-1$, 0, or 1 |
| `SQR(x)` | Square root |
| `RND(1)` | Pseudo-random number in $[0, 1)$ |
| `LOG(x)` | Natural logarithm |
| `EXP(x)` | $e^x$ |
| `SIN(x)` | Sine (radians) |
| `COS(x)` | Cosine (radians) |
| `TAN(x)` | Tangent (radians) |
| `ATN(x)` | Arctangent (radians) |
| `PI` | The constant $\pi \approx 3.14159$ |
| `TWOPI` | The constant $2\pi \approx 6.28318$ |
| `MAX(a,b)` | Larger of two values |
| `MIN(a,b)` | Smaller of two values |
| `PEEK(addr)` | Read a byte from memory address *addr* |
| `DEEK(addr)` | Read a 16-bit word from address *addr* |
| `FRE(0)` | Free BASIC program memory in bytes |
| `POS(0)` | Current cursor column position |

### 3.5.2 String Functions

| Function | Description |
|---|---|
| `LEN(s$)` | Number of characters in the string |
| `CHR$(n)` | Character whose ASCII code is $n$ |
| `ASC(s$)` | ASCII code of the first character |
| `STR$(n)` | Convert number to string |
| `VAL(s$)` | Convert string to number |
| `LEFT$(s$,n)` | Left $n$ characters |
| `RIGHT$(s$,n)` | Right $n$ characters |
| `MID$(s$,p,n)` | $n$ characters starting at position $p$ |
| `UCASE$(s$)` | Convert to uppercase |
| `LCASE$(s$)` | Convert to lowercase |
| `HEX$(n)` | Hexadecimal string representation of $n$ |
| `BIN$(n)` | Binary string representation of $n$ |

## 3.6 Memory Access

NovaBASIC gives you direct read and write access to the 6502 address space. This is useful for reading hardware registers, patching values, and interfacing with assembly routines.

`PEEK(addr)` Read one byte (0–255) from address *addr*.

`POKE addr, val` Write one byte to address *addr*.

`DEEK(addr)` Read a 16-bit little-endian word from *addr* and *addr*+1.

`DOKE addr, val` Write a 16-bit little-endian word.

`VARPTR(v)` Return the memory address of variable *v*. Useful when passing variable addresses to assembly routines.

```
10 REM read and print two bytes at $0300
20 PRINT PEEK(768); PEEK(769)
30 REM write a counter value
40 POKE 768, 42
50 PRINT PEEK(768)
```

**Warning**

Writing to the wrong address can crash the virtual machine or corrupt the BASIC interpreter. Know what you are writing to before you `POKE` into system areas. See the memory map in Appendix B for safe zones.

## 3.7 Style Tips

- **One idea per line.** Statements can be chained with colons (`:`), but a single clear statement per line is easier to read and debug.
- **Group by function.** Put constants and configuration near the top (lines 10–90), input routines in the 100s, game logic in the 200s–500s, display routines in the 600s, cleanup in the 900s.
- **Name things meaningfully.** `LIVES` is clearer than `L`; `SCORE` is clearer than `S1`. NovaBASIC accepts long names.
- **Define constants up front.** `10 MAXLIVES=5 : STARTSPEED=2` at the top of the program means you tune the game by changing one line, not hunting through hundreds.

## 3.8 Putting It Together

Enter and run this program. It builds an array of random values and prints them in indexed form.

```
10 DIM A(5)
20 FOR I=1 TO 5
30   A(I) = INT(RND(1) * 100)
40 NEXT I
50 FOR I=1 TO 5
60   PRINT "A(";I;") = ";A(I)
70 NEXT I
```

**Expected:** Five lines printed, each showing an index and a random integer between 0 and 99. The values differ every time you `RUN`.

Try modifying the program: change `100` to `10` to get single-digit values, or change `5` to `20` to see a longer array. The structure stays the same; only the constants change.

Chapter 4

# Graphics and Sprites

*"A blank screen is a canvas waiting to be claimed."*
— e6502 Virtual Computer Design Notes

NovaBASIC gives you direct access to a 320×200 pixel bitmap and a hardware sprite layer. Drawing commands operate on 16 colors; sprites add independently positioned, independently animated 16×16 objects on top of or behind the bitmap and text layers. This chapter covers the full graphics pipeline from mode selection to collision detection.

## 4.1 Display Modes

The virtual display has two independent layers: a text layer and a graphics bitmap layer. `MODE` selects how they are composited.

| Mode | Description |
|---|---|
| 0 | Text only. The graphics bitmap is not rendered. |
| 1 | Graphics over text. Bitmap is drawn on top of text characters. |
| 2 | Text over graphics. Text characters are drawn on top of the bitmap. |

The typical starting sequence for any graphics program is:

```
10 MODE 1
20 GCLS
30 GCOLOR 7
```

`MODE 1` activates pixel rendering. `GCLS` clears the bitmap to transparent (color 0). `GCOLOR` sets the active drawing color for all subsequent drawing commands.

Color 0 is transparent in the graphics layer. Setting a pixel to color 0 with `PLOT 0` or `FILL` erases it, letting the text layer or background show through. This is equivalent to `UNPLOT`.

To return to plain text output, switch back to `MODE 0`. You do not need to clear the bitmap when switching modes; the pixel data is preserved and will reappear if you switch back to `MODE 1` or `MODE 2`.

## 4.2 Drawing Commands

All drawing commands use the color set by `GCOLOR`. Coordinates must fall within the screen boundaries of X = 0–319 and Y = 0–199; pixels outside that range are silently clipped and no error is raised.

### Command reference

| Command | Description |
|---|---|
| `GCOLOR c` | Set the active drawing color. `c` is 0–15. If `c` = 0, NovaBASIC uses the current text foreground color instead of transparent. |
| `GCLS` | Clear the entire graphics bitmap to transparent (color 0). Does not affect the text layer. |
| `PLOT x,y` | Set the pixel at (`x`, `y`) to the current drawing color. |
| `UNPLOT x,y` | Set the pixel at (`x`, `y`) to transparent (color 0), effectively erasing it. |
| `LINE x0,y0,x1,y1` | Draw a straight line from (`x0`, `y0`) to (`x1`, `y1`) in the current drawing color. |
| `RECT x0,y0,x1,y1` | Draw a rectangle outline. (`x0`, `y0`) is the top-left corner; (`x1`, `y1`) is the bottom-right corner. |
| `FILL x0,y0,x1,y1` | Draw a solid filled rectangle using the same corner convention as `RECT`. |
| `CIRCLE cx,cy,r` | Draw a circle outline centered at (`cx`, `cy`) with radius `r` pixels. |
| `PAINT x,y` | Flood-fill from seed point (`x`, `y`), replacing all connected pixels of the same color with the current drawing color. |

### A drawing example

The following program draws a diagonal cross, a circle, and then fills the circle interior:

```
10 MODE 1 : GCLS
20 GCOLOR 9
30 LINE 0,0,319,199
40 LINE 319,0,0,199
50 GCOLOR 14
60 CIRCLE 160,100,60
70 GCOLOR 10
80 PAINT 160,100
90 VSYNC
```

Line 30–40 draws a white cross from corner to corner. Lines 60–80 add a yellow circle outline and then flood-fill the interior with green. `VSYNC` on line 90 holds the image for one frame before the program ends; without it the display may update before you see the result.

`PAINT` stops at pixel boundaries of a different color. Make sure the circle or region you want to fill has no gaps, otherwise the fill will leak out into the surrounding area. If in

doubt, draw the boundary in one step and fill immediately after.

## 4.3 Animation with VSYNC

The virtual display runs at 60 Hz. `VSYNC` suspends program execution until the start of the next video frame. One `VSYNC` call therefore consumes exactly one frame period (≈ 16.7 ms). This is the correct tool for controlling animation speed.

A minimal animation loop that moves a point across the screen:

```
10 MODE 1 : GCLS : GCOLOR 11
20 X = 0 : Y = 100
30 VSYNC
40 UNPLOT X, Y
50 X = X + 2
60 IF X > 319 THEN X = 0
70 PLOT X, Y
80 GOTO 30
```

The pattern is always: wait for VSYNC, erase the old position, update coordinates, draw the new position. Erasing before moving eliminates the ghost trail that builds up if you draw without erasing.

For smooth movement, do all erase operations for a frame, update all positions, and do all draw operations — all within a single VSYNC period. Never call `VSYNC` between the erase and redraw steps for the same object; that produces a one-frame flicker every cycle.

## 4.4 Sprites

Sprites are hardware-accelerated 16×16 pixel objects that move independently of the bitmap. NovaBASIC supports 16 sprites (indices 0–15), each with its own shape, position, priority, and flip state. Sprites do not modify the bitmap; they are composited at render time.

### Enabling and positioning sprites

A sprite must be enabled before it becomes visible:

```
10 SPRITE 0, ON
20 SPRITE 0, 160, 100
```

`SPRITE n,ON` activates sprite `n`. `SPRITE n,x,y` sets its screen position. Positions are in the same coordinate space as the bitmap (X = 0–319, Y = 0–199). Sprites may be positioned partially or fully off-screen; they are simply clipped without error.

To hide a sprite, use `SPRITE n,OFF`. This makes the sprite invisible without erasing its shape data. You can re-enable it later with `SPRITE n,ON` and it will reappear at its last recorded position.

### Sprite priority

Priority controls which layer a sprite is drawn on:

| Priority | Layer position |
|---|---|
| 0 | Behind all layers (below text and graphics) |
| 1 | Between the text and graphics layers |
| 2 | In front of all layers (above text and graphics) |

Priority is set via the MCP sprite tools when building shapes interactively; it can also be arranged by designing your program so that background sprites are enabled first and foreground sprites last.

### Defining sprite pixels with SPRITEDATA

Each sprite is 16 pixels wide by 16 pixels tall. Pixel data is loaded one row at a time using `SPRITEDATA`:

```
SPRITEDATA n, row, b1, b2, b3, b4, b5, b6, b7, b8
```

- `n` is the sprite index (0–15).
- `row` is the row to define (0–15, top to bottom).
- `b1` – `b8` are eight byte values (0–255).

Each byte encodes *two* pixels. The high nibble (upper four bits) is the left pixel; the low nibble (lower four bits) is the right pixel. Color 0 is transparent; colors 1–15 are the standard 16-color palette. With eight bytes per row and two pixels per byte, each row is exactly 16 pixels wide.

**Example:** a byte value of `$AC` (decimal 172) draws pixel color 10 (`A` in hex) on the left and pixel color 12 (`C` in hex) on the right. A value of `$00` leaves both pixels transparent.

The following example defines a simple 16×16 diamond shape in color 11 (cyan) and displays it:

```
10 REM DEFINE A DIAMOND SPRITE
20 SPRITEDATA 0,  0, 0,   0,   0,  0,  0,   0,   0,  0
30 SPRITEDATA 0,  1, 0,   0,   0, 11,  0,   0,   0,  0
40 SPRITEDATA 0,  2, 0,   0, 177,177,  0,   0,   0,  0
50 SPRITEDATA 0,  3, 0, 187, 187,187,187,   0,   0,  0
60 SPRITEDATA 0,  4, 0, 187, 187,187,187,   0,   0,  0
70 SPRITEDATA 0,  5, 0,   0, 177,177,  0,   0,   0,  0
80 SPRITEDATA 0,  6, 0,   0,   0, 11,  0,   0,   0,  0
90 SPRITEDATA 0,  7, 0,   0,   0,  0,  0,   0,   0,  0
100 REM ROWS 8-15 REMAIN TRANSPARENT (NO SPRITEDATA = NO CHANGE)
110 SPRITE 0, ON
120 SPRITE 0, 152, 92
130 VSYNC
```

Any row not explicitly defined by `SPRITEDATA` retains its previous pixel data. If you are reusing a sprite slot for a new shape, define all 16 rows (or clear the slot first). Rows you intentionally leave all-zero produce a fully transparent row.

The bytes in lines 30–80 use decimal notation. Working with hex notation is often more readable: `0xBB` = decimal 187, which encodes color 11 in both nibbles (solid cyan on both pixels of that byte). In NovaBASIC you can write hex literals directly in expressions using `&HBB` notation.

## 4.5 Sprite Collision Detection

NovaBASIC provides two collision functions that report when sprites overlap each other or touch non-transparent pixels on the background bitmap.

| Function | Returns |
|---|---|
| `COLLISION(n)` | Bitmask of other sprites currently overlapping sprite `n`. |
| `BUMPED(n)` | Bitmask indicating that sprite `n` has touched a non-transparent pixel in the graphics bitmap. |

Both functions return an integer bitmask. Bit `k` being set means sprite `k` is involved in the collision. For `COLLISION(n)`, if the result is non-zero then at least one other sprite overlaps sprite `n`; use `AND` with the appropriate bit to test for a specific sprite. For `BUMPED(n)`, a non-zero result means sprite `n` is touching a non-transparent pixel in the graphics bitmap.

**Warning**

Both collision registers clear automatically when read. Read each register exactly once per frame and store the result in a variable. If you call `COLLISION(n)` or `BUMPED(n)` a second time in the same frame you will get zero, missing collisions that occurred between reads.

A practical collision loop pattern:

```
100 VSYNC
110 C = COLLISION(0)
120 B = BUMPED(0)
130 IF C <> 0 THEN GOSUB 500
140 IF B <> 0 THEN GOSUB 600
150 REM UPDATE POSITIONS HERE
160 GOTO 100
```

Lines 110–120 read both registers once and store them. Lines 130–140 branch to handler routines only if a collision has occurred. All position updates happen after the collision check so that the same frame's register values are used consistently.

To test whether sprite `n` specifically collided with sprite 2, check bit 2 of the `COLLISION` result:

```
200 C = COLLISION(0)
210 IF (C AND 4) <> 0 THEN PRINT "HIT SPRITE 2"
```

## 4.6 Color Palette

NovaBASIC uses a fixed 16-color palette inspired by the Commodore 64. All graphics, text, sprite, background, and border colors use the same indices.

| Index | Color | Index | Color |
|---|---|---|---|
| 0 | Black | 8 | Orange |
| 1 | White | 9 | Brown |
| 2 | Red | 10 | Light Red |
| 3 | Cyan | 11 | Dark Grey |
| 4 | Purple | 12 | Medium Grey |
| 5 | Green | 13 | Light Green |
| 6 | Blue | 14 | Light Blue |
| 7 | Yellow | 15 | Light Grey |

The background color defaults to 0 (black), the text foreground to 1 (white), and the border to 6 (blue). Use `COLOR fg[,bg]` for text colors. The border color can be changed via `POKE 40973, c` where `c` is 0–15 ($A00D is the border color register).

## 4.7 The Copper (Raster Effects)

The copper is a per-pixel register-write system inspired by the Amiga's Copper coprocessor. It lets you change display registers at precise screen positions, enabling effects like color gradient backgrounds, split-screen modes, and parallax scrolling — all without any CPU involvement.

### How it works

A copper *program* is a list of up to 1024 events, each specifying:

- A screen position (X = 0–319, Y = 0–199).
- A target register.
- A value to write to that register.

When copper is enabled, the renderer checks for copper events at every pixel position. When it reaches a pixel that has a scheduled event, the register write fires immediately and affects all subsequent pixels on that frame.

### Writable registers

The copper can write to four VGC registers:

| Register | Address | Effect |
|---|---|---|
| RegMode | $A000 | Change display mode mid-screen |
| RegBgCol | $A001 | Change background color mid-screen |
| RegScrollX | $A005 | Shift horizontal scroll offset |
| RegScrollY | $A006 | Shift vertical scroll offset |

### Copper commands via POKE

The copper has no dedicated BASIC keyword. You control it by writing to the VGC command registers ($A010–$A016) and then writing the command byte to $A010.

| Code | Command |
|---|---|
| $1B | **Copper Add** — add an event. P0/P1 = X (16-bit), P2 = Y, P3/P4 = register (0–15 or $A000–$A00F), P5 = value. If an event at the same position for the same register already exists, the value is replaced. |
| $1C | **Copper Clear** — remove all copper events. |
| $1D | **Copper Enable** — start executing the copper program each frame. |
| $1E | **Copper Disable** — stop executing the copper program. |

### Example: background color gradient

The following program creates a vertical color gradient by scheduling a background color change at the start of each group of rows:

```
10 MODE 1 : GCLS
20 REM CLEAR ANY PREVIOUS COPPER PROGRAM
30 POKE $A010, $1C
40 REM ADD COLOR CHANGES AT DIFFERENT Y POSITIONS
50 FOR I = 0 TO 12
60   POKE $A011, 0 : POKE $A012, 0
70   POKE $A013, I * 15
80   POKE $A014, 1 : POKE $A015, 0
90   POKE $A016, I + 2
100  POKE $A010, $1B
110 NEXT I
120 REM ENABLE COPPER
130 POKE $A010, $1D
```

Line 30 clears any previous copper events. The loop on lines 50–110 adds 13 events, each changing the background color (register 1) at Y positions 0, 15, 30, . . . , 180 to successive color indices. Line 130 enables the copper. Each frame the background will display as a gradient of bands.

Copper events fire at per-pixel granularity. Setting X = 0 for all events produces clean horizontal bands. Setting different X values within the same row creates vertical split effects.

The maximum copper program size is 1024 events. Events are sorted by position automatically. Multiple events at the same position are applied in register-index order.

## 4.8 Compatibility Notes

**Warning**

The following commands and functions are parsed by NovaBASIC for source compatibility but currently have no effect at runtime:

- `SPRITESHAPE` — accepted without error, silently ignored.
- `SPRITECOLOR` — accepted without error, silently ignored.

- `SPRITEX(n)` — always returns 0; does not reflect actual sprite X position.
- `SPRITEY(n)` — always returns 0; does not reflect actual sprite Y position.

Programs that read sprite position back via `SPRITEX` / `SPRITEY` must instead track coordinates in their own variables.

## 4.9 Try It Now

Type and run the following program to see `MODE`, `GCLS`, `GCOLOR`, `RECT`, `CIRCLE`, and `PAINT` working together:

```
10 MODE 1 : GCLS : GCOLOR 10
20 RECT 10, 10, 309, 189
30 GCOLOR 14 : CIRCLE 160, 100, 50
40 GCOLOR 12 : PAINT 160, 100
```

Expected result: a green rectangle border frames the screen; inside it a yellow circle outline encloses a solid red filled region.

Try modifying `GCOLOR` values (1–15) and the `CIRCLE` radius to explore the coordinate system. Then add a second `CIRCLE` call on a new line and re-run to see both circles on the same canvas.

Chapter 5

# Sound and Music

*"Music is the arithmetic of sounds as optics is the geometry of light."*

— Claude Debussy

NovaBASIC includes a SID chip emulator — a software recreation of the MOS 6581 Sound Interface Device made famous by the Commodore 64. Three independent voices with four waveforms, ADSR envelopes, and a programmable filter deliver authentic chiptune sound. On top of the SID sits a three-voice MML music sequencer with per-frame effects including vibrato, portamento, arpeggios, pulse-width modulation, and filter sweeps.

This chapter covers every sound command from simple one-shot notes to full multi-voice compositions.

## 5.1 Quick-Start Overview

The sound system has three layers, each building on the one below:

1. **SOUND** — play a single note on the SID chip. Specify a MIDI note number, duration in frames, and an optional instrument preset.
2. **INSTRUMENT** — define a reusable preset that sets the SID waveform and ADSR envelope. Up to 16 presets (slots 0–15).
3. **MUSIC** — load MML (Music Macro Language) sequences into up to three voices and play them back with tempo, looping, and per-frame effects.

A minimal program that plays a note:

```
10 VOLUME 12
20 SOUND 60, 30
```

Line 10 sets master volume. Line 20 plays MIDI note 60 (middle C) for 30 frames (half a second at 60 Hz).

## 5.2 The SOUND Command

```
SOUND note, duration [, instrument]
```

- `note` — MIDI note number (0–127). Middle C is 60; A4 (concert pitch 440 Hz) is 69. See the MIDI note table below.

- `duration` — length in 1/60-second frames. A value of 60 plays for one second; 30 plays for half a second.
- `instrument` — optional instrument preset (0–15). If omitted, instrument 0 is used.

If `note` or `duration` is zero, the sound is stopped immediately.

`SOUND` triggers a one-shot sound effect through the music engine's SFX channel. It does not interrupt music playback; the engine allocates a voice for the effect and restores it when the sound completes.

### Common MIDI note numbers

| Note | MIDI | Approx. Frequency |
|---|---|---|
| C3 | 48 | 131 Hz |
| C4 (Middle C) | 60 | 262 Hz |
| D4 | 62 | 294 Hz |
| E4 | 64 | 330 Hz |
| F4 | 65 | 349 Hz |
| G4 | 67 | 392 Hz |
| A4 | 69 | 440 Hz |
| B4 | 71 | 494 Hz |
| C5 | 72 | 523 Hz |
| C6 | 84 | 1047 Hz |

The formula is: frequency $= 440 \times 2^{(midi-69)/12}$.

### A simple melody

```
10 VOLUME 12
20 DATA 60, 62, 64, 65, 67, 69, 71, 72
30 FOR N = 1 TO 8
40   READ M
50   SOUND M, 15
60   FOR I = 1 TO 15 : VSYNC : NEXT I
70 NEXT N
```

Each note plays for 15 frames (≈ 250 ms). The `VSYNC` loop holds the program for the same duration before the next note fires.

`SOUND` does not block program execution. Use a `VSYNC` loop after each `SOUND` call to create the gap between notes.

## 5.3 The INSTRUMENT Command

```
INSTRUMENT id, waveform, attack, decay, sustain, release
```

Defines a reusable sound preset in one of 16 instrument slots.

- `id` — slot number (0–15).

- `waveform` — SID waveform byte: `$10` = triangle, `$20` = sawtooth, `$40` = pulse, `$80` = noise.
- `attack` — attack rate (0–15). 0 is instantaneous; 15 is slowest.
- `decay` — decay rate (0–15). How quickly the volume drops from peak to the sustain level.
- `sustain` — sustain level (0–15). The steady-state volume held while the note plays. 15 = full volume; 0 = silent (percussive).
- `release` — release rate (0–15). How quickly the volume fades to silence after the note ends.

```
10 REM BRIGHT PULSE LEAD
20 INSTRUMENT 0, $40, 0, 9, 0, 6
30 REM WARM SAWTOOTH PAD
40 INSTRUMENT 1, $20, 4, 6, 12, 8
50 REM NOISE DRUM HIT
60 INSTRUMENT 2, $80, 0, 3, 0, 2
```

Instrument 0 is pre-initialized at boot with: pulse waveform (`$40`), attack 0, decay 9, sustain 0, release 6, pulse width 2048. All other slots (1–15) start as copies of slot 0.

### SID waveform reference

| Value | Name | Character |
|---|---|---|
| `$10` | Triangle | Soft and mellow; flute-like. Good for gentle melodies and background pads. |
| `$20` | Sawtooth | Buzzy and harmonically rich. Good for brass-like leads and bass lines. |
| `$40` | Pulse | Bold, hollow, classic chiptune sound. Pulse width can be modulated via MML for evolving timbres. |
| `$80` | Noise | Unpitched random output. Use for drums, hi-hats, explosions, and ambient textures. |

### ADSR envelope overview

The four parameters shape how a note's volume changes over time:

1. **Attack** ramps from silence to full amplitude.
2. **Decay** drops from full amplitude to the sustain level.
3. **Sustain** holds at a constant level while the note plays.
4. **Release** fades from the sustain level to silence after the note ends.

Sustain is a *level* (0–15); the other three are *rate* values where 0 is fastest and 15 is slowest. This matches the original SID chip behavior.

### Instrument recipes

| Sound | Wave | A | D | S | R | Notes |
|---|---|---|---|---|---|---|
| Chiptune lead | $40 | 0 | 9 | 0 | 6 | Sharp attack, no sustain |
| Warm pad | $20 | 8 | 6 | 12 | 10 | Slow fade-in, high sustain |
| Bass | $20 | 0 | 5 | 8 | 4 | Instant attack, medium body |
| Snare drum | $80 | 0 | 3 | 0 | 2 | Short noise burst |
| Hi-hat | $80 | 0 | 1 | 0 | 1 | Very short noise tick |
| Organ | $10 | 0 | 0 | 15 | 4 | Triangle at full sustain |
| Pluck | $40 | 0 | 12 | 0 | 8 | Fast decay, no sustain |

## 5.4 VOLUME

```
VOLUME level
```

Sets the SID master volume. `level` is 0–15 (only the low nibble is used). The default volume at boot is 12.

## 5.5 The MUSIC Engine

The music engine is a three-voice MML sequencer running on top of the SID chip. You write melodies and rhythms as text strings using Music Macro Language, load them into voices, and let the engine handle all the timing, instrument switching, and per-frame effects automatically.

### 5.5.1 Loading and Playing Sequences

```
MUSIC voice, "mml-string"
```

- `voice` — voice number 1–3.
- `"mml-string"` — an MML sequence (see Section 5.6).

Additional subcommands control playback:

| Command | Description |
|---|---|
| `MUSIC PLAY` | Start playback of all loaded voices. |
| `MUSIC STOP` | Stop playback and silence all music voices. |
| `MUSIC TEMPO bpm` | Set tempo in beats per minute. Default is 120. |
| `MUSIC LOOP ON` | Enable looping; voices restart when all finish. |
| `MUSIC LOOP OFF` | Disable looping (default). |
| `MUSIC PRIORITY v1[,v2[,v3]` | Set voice-stealing priority for sound effects. Lower-numbered voices are stolen first. Default: 3, 2, 1. |

### 5.5.2 A Complete Music Example

```
10 VOLUME 12
20 REM DEFINE INSTRUMENTS
30 INSTRUMENT 0, $40, 0, 9, 0, 6
40 INSTRUMENT 1, $20, 0, 5, 8, 4
```

```
50 REM LOAD VOICES
60 MUSIC 1, "T140 I0 L8 O4 CDEFGAB >C"
70 MUSIC 2, "T140 I1 L4 O3 C G C G"
80 REM START PLAYBACK
90 MUSIC LOOP ON
100 MUSIC PLAY
```

Line 60 loads a melody into voice 1: tempo 140, instrument 0, eighth notes, octave 4, ascending C major scale. Line 70 loads a bass line into voice 2: quarter notes, octave 3, alternating C and G. Line 100 starts playback; `MUSIC LOOP ON` on line 90 means the music repeats indefinitely.

### 5.5.3 Querying Music Status

Two functions let you check what the music engine is doing:

| Function | Returns |
|---|---|
| `PLAYING` | 1 if music is currently playing, 0 if stopped. |
| `MNOTE(voice)` | The MIDI note number currently sounding on `voice` (1–3), or 0 if that voice is silent. |

```
200 IF PLAYING THEN GOTO 200
210 PRINT "MUSIC FINISHED"
```

## 5.6 MML Reference

Music Macro Language (MML) is a compact text notation for music. Each voice receives its own MML string. The parser is case-insensitive; all input is converted to uppercase before processing.

### 5.6.1 Notes and Rests

| Syntax | Description |
|---|---|
| `C D E F G A B` | Play a note. Pitch is determined by the current octave. |
| `C#` or `C+` | Sharp (raise one semitone). |
| `C-` | Flat (lower one semitone). |
| `R` | Rest (silence for the note duration). |

### 5.6.2 Duration

A number following a note or rest sets its length as a note-value denominator:

| Denominator | Ticks | Name |
|---|---|---|
| 1 | 384 | Whole note |
| 2 | 192 | Half note |
| 4 | 96 | Quarter note |
| 8 | 48 | Eighth note |
| 16 | 24 | Sixteenth note |
| 32 | 12 | Thirty-second note |

Internally, one quarter note equals 96 ticks.

A dot ( `.` ) after the duration extends it by 50%: `C4.` plays for $96 \times 1.5 = 144$ ticks (dotted quarter).

If no duration is given, the default length set by `L` is used (initially a quarter note).

### 5.6.3 Ties

The ampersand ( `&` ) ties two durations together into a single sustained note:

`C4&8` → quarter + eighth = 144 ticks

Multiple ties can be chained: `C4&4&4` plays for $96 + 96 + 96 = 288$ ticks. A single `NoteOn` event is emitted with the combined duration.

### 5.6.4 Octave

| Cmd | Description |
|---|---|
| `On` | Set absolute octave (range 1–7; default 4). |
| `>` | Octave up (clamped to 7). |
| `<` | Octave down (clamped to 1). |

MIDI note calculation: `midi = (octave + 1) * 12 + semitone`, where C=0, D=2, E=4, F=5, G=7, A=9, B=11.

### 5.6.5 Default Length

`L8` sets the default note/rest duration to eighth notes. All subsequent notes and rests that omit an explicit duration will use this value.

### 5.6.6 Tempo

`T120` sets the tempo to 120 beats per minute. The default is 120. Tempo can appear anywhere in the MML string and takes effect immediately. At 120 BPM, one quarter note lasts exactly 0.5 seconds.

Tempo is global. If multiple voices contain `T` commands, the last one processed wins. It is best practice to set tempo in voice 1 only.

### 5.6.7 Instrument Selection

`I3` switches the current voice to instrument slot 3 (defined earlier with the `INSTRUMENT` BASIC command). Instrument changes take effect on the next note.

### 5.6.8 Loops

Square brackets repeat a section:

`[CDEF]3` → plays C D E F three times

The repeat count follows the closing bracket. If omitted, the default is 1 (no repetition). Loops do not nest.

### 5.6.9 Arpeggios

Curly braces define an arpeggio — a rapid cycling through multiple notes:

`{CEG}4` → cycle C, E, G at 60 Hz for one quarter note

Each frame advances to the next note in the list. Accidentals are supported inside the braces ( `{C#EG#}` ). The arpeggio duration follows the closing brace using standard duration syntax.

### 5.6.10 Per-Frame Effects

The music engine processes the following effects on every frame (60 Hz). These effects are set within MML and remain active until changed or a new note resets them.

*Vibrato*

`~6` sets vibrato depth to 6. Higher values produce wider pitch oscillation. `~0` turns vibrato off.

The vibrato oscillates at approximately 2.9 Hz (sine wave). The pitch offset is proportional to both the depth value and the current note frequency.

*Portamento (Pitch Slide)*

`/` before a note causes the voice to slide from the current pitch to the target note rather than jumping instantly.

`C4 /E4` → play C, then glide smoothly to E

The slide rate is approximately $\frac{1}{8}$ of the frequency distance per frame.

*Pulse Width*

`@P2048` sets the SID pulse width to 2048 (range 0–4095). This only affects the pulse waveform (`$40`). A value of 2048 gives a 50% duty cycle (square wave); lower or higher values create thinner, more nasal timbres.

*Pulse Width Modulation (PWM)*

`@PS+` starts sweeping the pulse width upward; `@PS-` sweeps downward; `@PS0` stops the sweep. The sweep rate is ±32 per frame, clamped to the 0–4095 range. PWM gives the pulse waveform a rich, evolving character.

```
60 MUSIC 1, "@P1024 @PS+ O4 L2 C E G >C"
```

*Filter Cutoff and Resonance*

`@F1024,8` sets the SID filter cutoff to 1024 (range 0–2047) and resonance to 8 (range 0–15). The resonance parameter is optional; if omitted, it defaults to 0.

*Filter Mode*

| Cmd | Mode |
|---|---|
| @FL | Low-pass (cuts highs, warm sound) |
| @FB | Band-pass (emphasizes a frequency band) |
| @FH | High-pass (cuts lows, thin sound) |
| @FO | Filter off |

*Filter Sweep*

`@FS+` sweeps the filter cutoff upward; `@FS-` sweeps downward; `@FS0` stops the sweep. The sweep rate is ±8 per frame, clamped to 0–2047.

```
60 MUSIC 1, "@FL @F200,12 @FS+ L4 O3 [CDEFGAB>C<]2"
```

This creates a classic filter sweep effect: low-pass filter starting at cutoff 200 with high resonance, sweeping upward through the melody.

### 5.6.11 MML Command Summary

| Command | Parameters | Description |
|---|---|---|
| A-G | [#/+/-][dur][.] | Play note |
| R | [dur][.] | Rest |
| O | 1–7 | Set octave |
| > | — | Octave up |
| < | — | Octave down |
| L | denominator | Default note length |
| T | bpm | Tempo (default 120) |
| I | 0–15 | Select instrument slot |
| & | [note]dur | Tie (extend note duration) |
| [...]n | repeat count | Loop section $n$ times |
| {notes} | [dur][.] | Arpeggio |
| ~ | depth (0=off) | Vibrato |
| / | — | Portamento (next note slides) |
| @P | 0–4095 | Set pulse width |
| @PS | +, -, 0 | PWM sweep direction |
| @F | cutoff[,res] | Filter cutoff (0–2047) and resonance (0–15) |
| @FL | — | Low-pass filter |
| @FB | — | Band-pass filter |
| @FH | — | High-pass filter |
| @FO | — | Filter off |
| @FS | +, -, 0 | Filter sweep direction |

Whitespace, tabs, newlines, and pipe characters ( `|` ) are ignored and can be used freely to format MML strings for readability.

## 5.7 SID File Playback

NovaBASIC can load and play standard `.sid` files — the native music format of the Commodore 64 scene:

| Command | Description |
|---|---|
| `SIDPLAY "filename" [, song]` | Load and play a `.sid` file. The optional `song` parameter selects which sub-tune to play (default 1). |
| `SIDSTOP` | Stop SID file playback. |

```
10 SIDPLAY "commando"
20 FOR I = 1 TO 600 : VSYNC : NEXT I
30 SIDSTOP
```

SID files are loaded from the `~/e6502-programs` directory. The `.sid` extension is added automatically. The SID player injects an IRQ trampoline into CPU RAM that calls the file's init and play routines at 60 Hz.

**Warning**

SID playback takes over the SID chip directly. `SOUND` and `MUSIC` commands will not produce audible output while a SID file is playing. Call `SIDSTOP` before using other sound commands.

## 5.8 Graphics File I/O

NovaBASIC can save and load VGC memory spaces to disk:

| Command | Description |
|---|---|
| `GSAVE "name", space, offset, len` | Save `len` bytes from VGC memory space starting at `offset` to a `.gfx` file. |
| `GLOAD "name", space, offset[, len]` | Load a `.gfx` file into VGC memory space at `offset`. If `len` is omitted, the entire file is loaded. |

VGC memory spaces:

| Space | Contents |
|---|---|
| 0 | Character RAM (2000 bytes) |
| 1 | Color RAM (2000 bytes) |
| 2 | Graphics bitmap (64000 bytes) |
| 3 | Sprite shape RAM (2048 bytes) |

```
10 MODE 1 : GCLS
20 GCOLOR 10 : CIRCLE 160, 100, 80
30 GSAVE "mycircle", 2, 0, 64000
40 REM LATER...
50 GLOAD "mycircle", 2, 0
```

## 5.9 Music Engine Architecture

For advanced users, understanding the engine internals helps write better music and avoid common pitfalls.

### Voice allocation

The music engine manages three music voices (mapped to SID voices 0–2) plus one shared SFX voice. When `SOUND` triggers a sound effect, the engine:

1. Looks for a voice with no music sequence loaded.
2. If all voices have sequences, steals a voice according to the priority order (default: voice 3 first, then 2, then 1).
3. Plays the SFX on the stolen voice; when done, restores the music voice.

### Timing

The engine ticks at 60 Hz. Tempo is converted to ticks per frame:

$$ticks per frame = \frac{96 \times BPM}{3600}$$

At 120 BPM this is 3.2 ticks per frame. A quarter note (96 ticks) takes exactly 30 frames = 0.5 seconds.

### Effect processing order

Each frame, active effects are processed in this order:

1. Arpeggio (cycle to next note)
2. PWM sweep (±32 per frame, clamped 0–4095)
3. Vibrato (sine wave at ~2.9 Hz)
4. Portamento (slide $\frac{1}{8}$ of remaining distance per frame)
5. Filter sweep (±8 per frame, clamped 0–2047)

## 5.10 Composition Tips

- Define all instruments before loading music sequences.
- Use `I` in MML to switch instruments mid-voice for timbral variety.
- Keep melody, harmony, and bass on separate voices. Each voice has its own instrument, octave, and effect state.
- Use `|` characters in MML strings as bar-line separators for readability: `"L8 CDEF | GABC"`.
- At 120 BPM: quarter = 30 frames, eighth = 15, sixteenth = 7.5. Use `T` to control tempo rather than adjusting note lengths.
- Use `@PS+` and `@PS-` on pulse waveforms for rich, evolving textures.
- Combine `@FL` with `@FS+` for classic acid-bass filter sweeps.
- The noise waveform (`$80`) with short ADSR makes convincing drums. Load a noise instrument and trigger it with `SOUND` while the music plays.

## 5.11 Deprecated Commands

**Warning**

The following commands from earlier versions have been superseded:

- `WAVE` — raises a syntax error. Use `INSTRUMENT` instead.
- `ENVELOPE` — replaced by `INSTRUMENT`. Programs should be updated to use the new six-parameter syntax.

## 5.12 Try It Now

Type and run the following program to hear a three-voice arrangement with instrument presets, MML sequences, filter effects, and looping:

```
10 VOLUME 12
20 INSTRUMENT 0, $40, 0, 9, 0, 6
30 INSTRUMENT 1, $20, 0, 5, 8, 4
40 INSTRUMENT 2, $80, 0, 3, 0, 2
50 MUSIC 1, "T120 I0 L8 O4 CDEFGAB >C2"
60 MUSIC 2, "T120 I1 L4 O3 C G C G"
70 MUSIC 3, "T120 I2 L8 O4 R C R C R C R C"
80 MUSIC LOOP ON
90 MUSIC PLAY
```

Expected result: a three-voice loop with a pulse-wave melody, sawtooth bass, and noise percussion. The music repeats until you type `MUSIC STOP` in direct mode.

Experiments:

- Change `T120` to `T180` for a faster tempo.
- Add vibrato to voice 1: change the MML to start with `"T120 I0 ~4 L8 O4 ..."`.
- Add a filter sweep to voice 2: `"T120 I1 @FL @F200,10 @FS+ L4 O3 C G C G"`.

Chapter 6

# Expansion Memory

> *"The problem with a 64-kilobyte address space is not that it is small.*
> *It is that everything you want to do next is just barely larger than it."*

The 6502 CPU can directly address 64 KB. That is enough for code, stack, screen RAM, and a modest program, but it leaves little room for data-heavy work: large sprite sheets, level maps, sample tables, or pre-rendered buffers quickly overflow the available RAM. The NovaBASIC expansion memory system solves this by providing an additional 512 KB of RAM that lives outside the CPU address space, accessible through a set of commands and a window-mapping mechanism.

## 6.1 Why Expansion Memory?

The CPU address bus is 16 bits wide. After subtracting ROM, the VGC, the sound controller, the file I/O coprocessor, the XMC registers, and screen RAM, your usable BASIC RAM runs from `$0280` to `$9FFF` — approximately 39 KB. That is enough for programs and variables, but becomes tight the moment you try to cache a full screen buffer, store multiple music tracks, or keep several sprite-sheet layers in memory simultaneously.

Expansion memory (**XRAM**) gives you up to 512 KB of additional storage with no impact on the CPU address space. You cannot execute code from XRAM directly, but you can:

- Store and retrieve arbitrary data blocks by raw offset.
- Name blocks and recall them without tracking raw addresses.
- Map 256-byte XRAM pages directly into CPU address windows for transparent byte-level access via `PEEK` and `POKE`.
- Allocate and free regions using a lightweight page-tracking allocator.

The XRAM hardware is controlled by the Expansion Memory Controller (XMC), mapped at `$BA00`. You normally never touch those registers directly — the BASIC commands handle all of it for you.

## 6.2 Banks and Status

XRAM is divided into **banks**, each 64 KB in size. With the default 512 KB configuration there are **8 banks** (numbered 0 through 7). One bank is always *active*; raw-offset commands operate within that bank.

### Checking memory status: `XMEM`

Type `XMEM` at the prompt to print a status summary:

```
XMEM
8 BANKS, 512 KB XRAM, BANK 0, USED 0, FREE 2048 PAGES
```

The output shows total bank count, total XRAM capacity, the currently active bank, and the number of 256-byte pages that are in use versus free across the entire 512 KB space. Two thousand and forty-eight free pages means 512 KB is available.

### Selecting a bank: `XBANK n`

```
XBANK 3
```

Selects bank 3 as the active bank. Subsequent `XPOKE`, `XPEEK`, and raw `STASH`/`FETCH` operations use offsets within that bank. Valid range is 0 to 7 for the default 512 KB configuration.

Named blocks (`STASH "name"` / `FETCH "name"`) are allocated automatically across the full XRAM space and are not limited to the active bank. The active bank setting only affects raw-offset operations.

## 6.3 Single-Byte Access

For simple tasks — writing a flag, reading a configuration byte, probing a value — the single-byte commands are the most direct interface.

#### `XPOKE offset,value`

Writes *value* (0–255) to *offset* within the currently active bank. Offsets run from 0 to 65535.

```
XBANK 0
XPOKE 0,42
XPOKE 1,255
```

#### `XPEEK(offset)`

Reads a single byte from *offset* in the active bank and returns it as a numeric value. You can use it anywhere a number is valid:

```
PRINT XPEEK(0)
V = XPEEK(1) + XPEEK(2)
```

`XPOKE` and `XPEEK` are convenient for a handful of bytes or for inspecting specific locations. For moving blocks of data use `STASH` and `FETCH`, which issue a single hardware transfer rather than looping byte by byte.

## 6.4 Bulk Transfers

Moving data in and out of XRAM is the core workflow for anything non-trivial. The raw bulk-transfer commands operate on explicit offsets within the active bank.

### Raw `STASH` — CPU RAM to XRAM

```
STASH ramaddr,xramoffset,length
```

Copies *length* bytes starting at CPU address *ramaddr* into XRAM starting at *xramoffset* within the active bank.

### Raw `FETCH` — XRAM to CPU RAM

```
FETCH ramaddr,xramoffset,length
```

Copies *length* bytes from XRAM at *xramoffset* back to CPU RAM starting at *ramaddr*. The destination must be within writable RAM (`$0000`–`$BFFF`).

### Example: saving and restoring a character screen buffer

The character screen RAM occupies `$AA00`–`$B1CF` (2000 bytes). You can snapshot it to XRAM bank 0 at offset 0 and restore it later:

```
10 REM Save character screen to XRAM
20 XBANK 0
30 STASH 43520,0,2000
40 PRINT "SCREEN STASHED."
50 REM ...  do other things ...
60 REM Restore character screen from XRAM
70 FETCH 43520,0,2000
80 PRINT "SCREEN RESTORED."
```

`$AA00` = 43520 decimal. The color RAM at `$B1D0` is an additional 2000 bytes; stash it at offset 2000 in the same bank to capture the full display state including colors.

## 6.5 Named Blocks

Raw offsets require you to track where each block lives. The named-block interface lets you store data under a string name and retrieve it without knowing its physical XRAM address.

### Storing a named block: `STASH "name",ramaddr,length`

```
STASH "MYDATA",2048,16
```

Allocates a region in XRAM, copies 16 bytes from CPU address 2048 into it, and registers the block under the name `MYDATA`. Names are 1 to 28 characters and are case-insensitive. If a block with that name already exists and is large enough for the new data, it is overwritten in place. If the existing block is too small, it is freed and a new one is allocated automatically.

### Loading a named block: `FETCH "name",ramaddr`

```
FETCH "MYDATA",2048
```

Finds the named block and copies its full contents back to CPU RAM at *ramaddr*. No length argument is required; the XMC remembers the exact size.

### Listing named blocks: `XDIR`

```
XDIR
```

Prints all named blocks sorted alphabetically (case-insensitive), showing each name and its stored size in bytes.

### Deleting a named block: `XDEL "name"`

```
XDEL "MYDATA"
```

Removes the named block and releases its XRAM pages back to the free pool.

### Complete example: named-block round-trip

```
10 FOR I=0 TO 15:POKE 2048+I,I+10:NEXT I
20 STASH "MYDATA",2048,16
30 FOR I=0 TO 15:POKE 2048+I,0:NEXT I
40 FETCH "MYDATA",2048
50 FOR I=0 TO 15:PRINT PEEK(2048+I);:NEXT I
60 XDEL "MYDATA"
```

Line 10 writes values 10 through 25 into RAM at address 2048. Line 20 stashes them to XRAM under the name `MYDATA`. Line 30 zeroes the same RAM region to confirm the data is no longer in CPU RAM. Line 40 fetches the block back. Line 50 prints the restored values — you should see `10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25`. Line 60 cleans up.

## 6.6 Low-Level Allocation

For programs that need to manage XRAM regions dynamically without names, two lower-level commands are available.

### `XALLOC length`

Allocates *length* bytes in XRAM and returns a numeric handle (1–255). The handle identifies the allocated region. You are responsible for recording it; the XMC does not associate a name.

```
H = XALLOC(4096)
PRINT "HANDLE: ";H
```

`XFREE offset,length`

Releases the XRAM range starting at *offset* with the given *length*. Any named or unnamed blocks that overlap that range are removed from the allocation table.

**Warning**

**XRESET** clears all allocation tracking in one step — named blocks, unnamed allocations, and page usage records are all discarded and cannot be recovered. The raw bytes in XRAM are not erased, but every name and handle is gone. Use `XRESET` only when you want a clean slate, such as at program startup.

## 6.7 Memory Windows

For the highest-performance XRAM access — or for assembly code that needs to use ordinary load and store instructions against XRAM — you can map an XRAM page directly into the CPU address space using a **window**.

There are four windows, each exactly 256 bytes wide, at fixed CPU addresses:

| Window | CPU Address Range | Size |
|---|---|---|
| 0 | $BC00–$BCFF | 256 bytes |
| 1 | $BD00–$BDFF | 256 bytes |
| 2 | $BE00–$BEFF | 256 bytes |
| 3 | $BF00–$BFFF | 256 bytes |

When a window is mapped, any `PEEK` or `POKE` to its CPU address range reads or writes directly into XRAM. No transfer command is needed.

`XMAP window,offset`

Maps the 256-byte XRAM page that contains *offset* into the specified window (0–3). The offset is rounded down to a 256-byte page boundary automatically.

```
10 REM Map XRAM page 0 into window 0 at $BC00
20 XMAP 0,0
30 FOR I=0 TO 255
40 POKE 48128+I,I
50 NEXT I
60 PRINT PEEK(48128);" ";PEEK(48255)
```

`$BC00` = 48128 decimal. After `XMAP 0,0`, writes to `$BC00`–`$BCFF` go directly into XRAM. The `PRINT` on line 60 should show `0 255`.

`XUNMAP window`

Unmaps the specified window. Reads and writes to that CPU address range no longer reach XRAM.

```
XUNMAP 0
```

**Warning**

Mapped windows are shared address space visible to both BASIC and any running assembly code. If two windows are mapped to the same XRAM page, or if BASIC and

assembly both access the same window, coordinate ownership carefully to avoid corruption. Unmap windows you are finished with.

## 6.8 Error Codes

When an XRAM command fails, NovaBASIC reports an error derived from the XMC error register. The complete set:

| Code | Meaning |
|---|---|
| 0 | No error |
| 1 | Address out of range |
| 2 | Bad arguments |
| 3 | Named block not found |
| 4 | No space available |
| 5 | Invalid name |
| 6 | End of directory |

## 6.9 Try It Now

**Named block round-trip**

```
10 FOR I=0 TO 15:POKE 2048+I,I+10:NEXT I
20 STASH "MYDATA",2048,16
30 FOR I=0 TO 15:POKE 2048+I,0:NEXT I
40 FETCH "MYDATA",2048
50 FOR I=0 TO 15:PRINT PEEK(2048+I);:NEXT I
60 XDEL "MYDATA"
```

Expected result: line 50 prints `10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25`, confirming the data survived the round-trip through XRAM and back.

Chapter 7

# Assembly and Special Chips

> *"To understand what the machine is actually doing, you have to speak its language — and that language is always closer to the metal than any other abstraction on top of it."*

NovaBASIC is built on a 6502 core and runs on top of a set of memory-mapped coprocessors. Most programs never need to touch hardware registers directly, but knowing where everything lives gives you the full picture: why certain address ranges are reserved, what BASIC commands actually do at the hardware level, and how to write assembly routines that cooperate cleanly with the BASIC runtime.

## 7.1 The Memory Map

The full 64 KB address space is partitioned as follows:

| Range | Size | Purpose | Access |
|---|---|---|---|
| `$0000`–`$00FF` | 256 B | Zero Page | R/W |
| `$0100`–`$01FF` | 256 B | Stack | R/W |
| `$0200`–`$027F` | 128 B | System Vectors | R/W |
| `$0280`–`$9FFF` | 39 KB | BASIC RAM | R/W |
| `$A000`–`$A01E` | 31 B | VGC Registers | R/W |
| `$A100`–`$A1FF` | 256 B | Sound Controller (VSC) | R/W |
| `$AA00`–`$B1CF` | 2000 B | Character RAM (80×25) | R/W |
| `$B1D0`–`$B99F` | 2000 B | Color RAM (80×25) | R/W |
| `$B9A0`–`$B9EF` | 80 B | File I/O Controller (FIO) | R/W |
| `$BA00`–`$BA3F` | 64 B | Expansion Memory Controller (XMC) | R/W |
| `$BC00`–`$BFFF` | 1024 B | XRAM Windows (when mapped) | R/W |
| `$C000`–`$FFFF` | 16 KB | ROM (NovaBASIC) | R only |

Everything from `$A000` upward through `$BFFF` is hardware I/O or managed window space. Writing to ROM (`$C000`+) has no effect.

### System Vectors at `$0200`

At boot, NovaBASIC initializes a vector table at page `$02` with the base addresses of each hardware controller. Assembly code can read these rather than hard-coding addresses, so programs remain compatible if the memory map is adjusted in a future ROM version:

| Address | Value | Meaning |
|---|---|---|
| $0200–$0201 | $A000 | VGC base |
| $0202–$0203 | $A010 | VGC command register |
| $0204–$0205 | $AA00 | Character RAM base |
| $0206–$0207 | $B1D0 | Color RAM base |
| $0208–$0209 | $A100 | VSC base |
| $020A–$020B | $B9A0 | FIO base |
| $020C–$020D | $BA00 | XMC base |

Each entry is a 16-bit little-endian address stored at the indicated pair.

## 7.2 Talking to Hardware from BASIC

Because the hardware controllers are memory-mapped, you can read and write their registers with ordinary `PEEK` and `POKE` calls from BASIC. This is the simplest way to experiment with hardware state or build lightweight diagnostic tools.

### Reading the frame counter

The VGC increments a frame counter register at `$A008` on every display frame. Reading it gives you a running frame tick useful for timing and animation:

```
PRINT PEEK(40968)
```

`$A008` = 40968 decimal. The counter wraps from 255 to 0 (it is an 8-bit register).

### Checking the active sprite count

```
PRINT "SPRITES: ";PEEK(40969)
```

`$A009` = 40969 decimal. This read-only register holds the number of currently enabled sprites.

### Reading keyboard input

```
K = PEEK(40975)
```

`$A00F` = 40975 decimal. This is the VGC character input register. Reading it returns the last character received, or 0 if none. BASIC's own `INKEY$` command uses the same register.

### Character output

Writing a character code to `$A00E` (40974 decimal) emits that character to the current cursor position, exactly as BASIC's `PRINT` does internally:

```
POKE 40974,65
REM prints the letter A
```

## 7.3 The CALL and USR Interface

NovaBASIC provides two ways to execute machine code from within a BASIC program.

`CALL addr`

Performs a `JSR` to *addr*. Execution resumes in BASIC after the machine-code routine executes an `RTS`. There is no parameter passing; `CALL` is a simple subroutine jump. You are responsible for preserving CPU registers if the routine will return to BASIC in a clean state.

```
10 CALL 49152
REM jumps to machine code at $C000
```

`USR(x)`

Calls a user-defined machine-code routine, passing the numeric value $x$ through the 6502 floating-point accumulator (FAC). The routine can read, modify, and return a value through the same register. `USR(x)` is a numeric function and its result can be used in an expression:

```
10 V = USR(42)
20 PRINT "RESULT: ";V
```

The address of the USR routine is set by storing a 16-bit pointer in the appropriate zero-page location before calling. Consult the EhBASIC 2.22 documentation for the exact zero-page addresses used by the USR vector.

## 7.4 Interrupts

The 6502 supports two interrupt lines: the maskable IRQ and the non-maskable NMI. NovaBASIC lets you handle both from within a BASIC program, which is useful for writing interrupt-driven input handlers, timing routines, and co-operative multitasking sketches.

### Setting up a handler: `IRQ linenumber` and `NMI linenumber`

```
10 IRQ 1000
20 NMI 2000
```

When an IRQ fires, execution branches to line 1000. When an NMI fires, execution branches to line 2000. The handlers are ordinary BASIC subroutines.

### Returning from a handler: `RETIRQ` and `RETNMI`

The last statement in an IRQ handler must be `RETIRQ`; the last statement in an NMI handler must be `RETNMI`. These are not interchangeable with `RETURN` — they restore the correct CPU state and re-enable the interrupt flag.

```
1000 REM IRQ handler
1010 PRINT "IRQ FIRED"
1020 RETIRQ

2000 REM NMI handler
2010 PRINT "NMI FIRED"
2020 RETNMI
```

**Warning**

Interrupt handlers run in the context of the BASIC interpreter. Keep them short. Avoid file I/O, heavy computation, or anything that re-enters the interpreter in an unexpected state. Long handlers can cause instability.

## 7.5 XMC Assembly Helpers

The NovaBASIC ROM exports a set of labelled helper routines for accessing the XMC from assembly code. Using these helpers instead of writing to XMC registers directly keeps your code clean and gives you error detection for free.

All helpers follow the same convention: on return, **carry clear** means success and **carry set** means an error occurred, with the XMC error code in the accumulator.

| Label | Purpose |
|---|---|
| `LAB_XM_SETADDR` | Set the 24-bit XRAM address: A = low byte, X = mid byte, Y = high byte. |
| `LAB_XM_STATUS` | Read status snapshot: A = status register, X = error code. |
| `LAB_XM_GETBYTE` | Read byte at current XADDR: A = value on success. |
| `LAB_XM_PUTBYTE` | Write byte at current XADDR: A = value to write. |
| `LAB_XM_STASH` | Bulk copy RAM to XRAM (preload XMC_RAML/H, XMC_LENL/H). |
| `LAB_XM_FETCH` | Bulk copy XRAM to RAM (preload XMC_RAML/H, XMC_LENL/H). |
| `LAB_XM_FILL` | Fill XRAM range with a byte value (preload XMC_DATA, XMC_LENL/H). |
| `LAB_XM_ALLOC` | Allocate a block: preload XMC_LENL/H; XADDR and handle returned in registers. |

### Example: reading one byte from XRAM in assembly

```
; Set 24-bit address 0x010000 (bank 1, offset 0)
LDA #$00
LDX #$00
LDY #$01
JSR LAB_XM_SETADDR

JSR LAB_XM_GETBYTE
BCS error
; A now holds the byte value
```

The ROM helper labels are defined in the NovaBASIC assembly source (`ehbasic/basic.asm`). If you assemble custom ROM extensions or overlays, link against the same symbol file to pick up these addresses.

## 7.6 VGC Register-Level Programming

The VGC command pipeline works by writing parameters to registers `$A011`–`$A01E`, then writing the command byte to `$A010`. The VGC executes the command synchronously. This is exactly what every graphics BASIC command does under the hood.

From assembly you can issue any VGC command directly:

```
; Issue GCLS command (clear graphics layer)
LDA #$07
STA $A010
```

The full VGC command code reference, including copper commands ($1B–$1E) and memory I/O commands ($19–$1A), is documented in Appendix B.

## 7.7 SID Chip Access

The SID chip registers at $D400–$D41C are write-intercepted within the ROM address range. Assembly code can write to them directly:

```
; Set voice 0 to sawtooth waveform, gate on
LDA #$21
STA $D404
```

The SID register layout matches the MOS 6581. Per-voice registers occupy 7 bytes each (voice 0 at $D400, voice 1 at $D407, voice 2 at $D40E). Filter and volume registers are at $D415–$D418. See Appendix B for the full register map.

**Warning**

The BASIC `INSTRUMENT`, `SOUND`, and `MUSIC` commands manage SID registers automatically. Direct SID register writes from assembly will conflict with the music engine unless you stop all music and SFX first.

## 7.8 Copper Programming from Assembly

The copper system is controlled through VGC command registers. From assembly, write parameters to $A011–$A016, then write the command code to $A010:

```
; Add copper event:  at Y=50, X=0, set BgCol to color 5
LDA #$00 :  STA $A011
LDA #$00 :  STA $A012
LDA #50 :  STA $A013
LDA #$01 :  STA $A014
LDA #$00 :  STA $A015
LDA #$05 :  STA $A016
LDA #$1B : STA $A010

; Enable copper
LDA #$1D : STA $A010
```

See the Copper section in Chapter 4 for a full explanation of writable registers and programming patterns.

## 7.9 Try It Now

**Read the frame counter via PEEK**

```
10 REM Read frame counter via PEEK
20 FOR I=1 TO 60:VSYNC:NEXT I
30 PRINT "FRAMES: ";PEEK(40968)
```

`$A008` = 40968 decimal. After 60 `VSYNC` waits (approximately one second at 60 Hz), the frame counter register reflects the elapsed frame ticks. The value will be somewhere in the range 0–255 because the counter wraps after 256 frames. Run the program several times and observe how the value changes.

Chapter A

# Command Reference

## A.1 NovaBASIC Command Quick Reference

The tables below list every NovaBASIC statement in syntax order. Arguments shown in `[brackets]` are optional. The `LET` keyword is always optional before an assignment.

### Program Control

| Syntax | Purpose |
|---|---|
| `RUN` | Execute program from the lowest line number. |
| `LIST [start[-end]]` | Display program lines; omit range to list all. |
| `NEW` | Clear program and all variables from memory. |
| `CONT` | Continue execution after `STOP` or Ctrl-C. |
| `END` | Terminate program and return to direct mode. |
| `STOP` | Break execution; `CONT` resumes at next statement. |
| `CLEAR` | Clear all variables and arrays; program is retained. |

### Flow Control

| Syntax | Purpose |
|---|---|
| `GOTO line` | Jump unconditionally to a line number. |
| `GOSUB line` | Call subroutine at line; return address is stacked. |
| `RETURN` | Return from the most recent `GOSUB`. |
| `FOR var=start TO end [STEP n]` | Begin a counted loop; step defaults to 1. |
| `NEXT var` | Advance loop variable and branch back if not done. |
| `IF expr THEN ... [ELSE ...]` | Conditional execution; `ELSE` is optional. |
| `ON expr GOTO l1,l2,...` | Branch to the *n*th line in the list. |
| `ON expr GOSUB l1,l2,...` | Call the *n*th subroutine in the list. |
| `DO` | Begin an indefinite loop body. |
| `LOOP [WHILE expr]` | End loop; re-enter while *expr* is true. |
| `LOOP [UNTIL expr]` | End loop; re-enter until *expr* becomes true. |

### Variables and Data

| Syntax | Purpose |
|---|---|
| `LET var=expr` | Assign value to variable ( `LET` is optional). |
| `DIM var(size)` | Declare a single-dimension array. |
| `DATA val,val,...` | Embed constant values for `READ` . |
| `READ var,var,...` | Read successive values from `DATA` statements. |
| `RESTORE [line]` | Reset the `DATA` pointer, optionally to a line. |
| `SWAP var1,var2` | Exchange the values of two variables. |
| `INC var` | Increment a numeric variable by 1. |
| `DEC var` | Decrement a numeric variable by 1. |

### Input / Output

| Syntax | Purpose |
|---|---|
| `PRINT expr[;expr...]` | Output values to screen; `;` suppresses newline. |
| `INPUT ["prompt";]var` | Display optional prompt and read from keyboard. |
| `GET var` | Read a single keypress into variable (non-blocking). |
| `WIDTH n` | Set the output line width in characters. |

### Memory

| Syntax | Purpose |
|---|---|
| `POKE addr,val` | Write an 8-bit byte to a 6502 address. |
| `DOKE addr,val` | Write a 16-bit word (little-endian) to a 6502 address. |
| `PEEK(addr)` | Read an 8-bit byte from a 6502 address (function). |
| `DEEK(addr)` | Read a 16-bit word (little-endian) from a 6502 address (function). |

### Bit Operations

| Syntax | Purpose |
|---|---|
| `BITSET var,bit` | Set bit $n$ (0-based) in an integer variable. |
| `BITCLR var,bit` | Clear bit $n$ in an integer variable. |
| `BITTST(val,bit)` | Return 1 if bit $n$ is set in *val*, else 0. |

### File Operations

| Syntax | Purpose |
|---|---|
| `SAVE "name"` | Save the current BASIC program to disk as `name.bas`. |
| `LOAD "name"` | Load a previously saved program from disk. |
| `DIR` | List all saved `.bas` programs. |
| `DEL "name"` | Delete a saved program from disk. |

### Text Display

| Syntax | Purpose |
|---|---|
| `CLS` | Clear the text screen and home the cursor. |
| `COLOR fg[,bg]` | Set foreground and optional background color (0–15). |
| `LOCATE x,y` | Move the text cursor to column $x$ (0–79), row $y$ (0–24). |

### Graphics

The graphics layer is 320×200 pixels. Colors are indices 0–15; color 0 is transparent on the graphics layer.

| Syntax | Purpose |
|---|---|
| `MODE n` | Display mode: 0=text only, 1=graphics over text, 2=text over graphics. |
| `GCLS` | Clear the graphics layer to transparent (all pixels 0). |
| `GCOLOR c` | Set the graphics draw color (0–15; only the low nibble is used). |
| `PLOT x,y` | Set pixel at $(x,y)$ to the current draw color. |
| `UNPLOT x,y` | Clear pixel at $(x,y)$ to 0 (transparent). |
| `LINE x0,y0,x1,y1` | Draw a straight line between two points. |
| `RECT x0,y0,x1,y1` | Draw a rectangle outline. |
| `FILL x0,y0,x1,y1` | Draw a filled solid rectangle. |
| `CIRCLE cx,cy,r` | Draw a circle outline centred at $(cx,cy)$. |
| `PAINT x,y` | Flood-fill from seed point $(x,y)$. |
| `VSYNC` | Wait for the next video frame boundary (60 Hz). |

### Sprites

Sprites are 16×16 pixels, 4-bit multicolor. Up to 16 sprites are available (indices 0–15).

| Syntax | Purpose |
|---|---|
| `SPRITE n,ON` | Enable (show) sprite $n$. |
| `SPRITE n,OFF` | Disable (hide) sprite $n$. |
| `SPRITE n,x,y` | Set sprite $n$ screen position. |
| `SPRITEDATA n,row,b1...b8` | Define one row of sprite shape data (8 bytes, 16 pixels). |

`SPRITESHAPE` and `SPRITECOLOR` are recognized tokens (for source compatibility) but their ROM handlers currently perform no hardware action.

### Sound and Music

The SID chip provides 3 voices. The music engine adds a three-voice MML sequencer with instrument presets, tempo, looping, and per-frame effects.

| Syntax | Purpose |
|---|---|
| `SOUND note,dur[,inst]` | Play MIDI *note* (0–127) for *dur* frames (1/60 s). Optional *inst* selects instrument preset (0–15, default 0). |
| `VOLUME level` | Set SID master volume (0–15; low nibble only). |
| `INSTRUMENT id,wave,a,d,s,r` | Define instrument preset *id* (0–15). *wave*: $10=tri, $20=saw, $40=pulse, $80=noise. *a,d,s,r*: ADSR values 0–15. |
| `MUSIC voice,"mml"` | Load MML sequence into *voice* (1–3). |
| `MUSIC PLAY` | Start music playback. |
| `MUSIC STOP` | Stop music and silence all music voices. |
| `MUSIC TEMPO bpm` | Set playback tempo (default 120). |
| `MUSIC LOOP ON` | Enable looping (restart when all voices finish). |
| `MUSIC LOOP OFF` | Disable looping (default). |
| `MUSIC PRIORITY v1[,v2[,v3]` | Set voice-stealing priority for SFX. |
| `SIDPLAY "name"[,song]` | Play a `.sid` file; optional *song* number (default 1). |
| `SIDSTOP` | Stop SID file playback. |

Warning

`WAVE` is deprecated and raises a syntax error. Use `INSTRUMENT` to set waveform and ADSR in a single command.

### Graphics File I/O

| Syntax | Purpose |
|---|---|
| `GSAVE "name",space,offset,len` | Save VGC memory to a `.gfx` file. *space*: 0=screen, 1=color, 2=gfx bitmap, 3=sprite shapes. |
| `GLOAD "name",space,offset[,len]` | Load a `.gfx` file into VGC memory. If *len* is omitted, loads the entire file. |

### Expansion Memory

XRAM is banked memory outside the 6502 address space, accessed via the XMC coprocessor. See Chapter 6 for a full programming guide.

| Syntax | Purpose |
|---|---|
| `XMEM` | Print XRAM bank count and page usage statistics. |
| `XBANK n` | Select the active 64 KB XRAM bank; *n* must be < total banks. |
| `XPOKE offset,value` | Write one byte to XRAM at the given offset in the active bank. |
| `XPEEK(offset)` | Read one byte from XRAM at the given offset (function). |
| `STASH ram,offset,length` | Copy *length* bytes from CPU RAM to XRAM (raw, no name). |
| `FETCH ram,offset,length` | Copy *length* bytes from XRAM to CPU RAM (raw). |
| `STASH "name",ram,length` | Store a named XRAM block from CPU RAM. |
| `FETCH "name",ram` | Load a named XRAM block back into CPU RAM. |
| `XDIR` | List all named XRAM blocks and their sizes. |
| `XDEL "name"` | Delete a named XRAM block. |
| `XALLOC length` | Allocate an unnamed XRAM block; returns offset in `XPEEK` result. |
| `XFREE offset,length` | Release a raw XRAM range from usage tracking. |
| `XRESET` | Clear all XRAM allocation and named-block state (destructive). |
| `XMAP window,offset` | Map an XRAM page to CPU window 0–3 (addresses $BC00–$BFFF). |
| `XUNMAP window` | Unmap a CPU window (0–3). |

**Warning**

`XRESET` destroys all named-block metadata and usage-tracking information. The raw XRAM contents are not zeroed, but all allocation records are lost.

### Interrupts and Machine Code

| Syntax | Purpose |
|---|---|
| `CALL addr` | Execute machine code subroutine at 6502 address; `JSR` / `RTS` pair. |
| `IRQ line` | Redirect the IRQ vector to a BASIC line number handler. |
| `NMI line` | Redirect the NMI vector to a BASIC line number handler. |
| `RETIRQ` | Return from an IRQ handler (re-enables interrupts). |
| `RETNMI` | Return from an NMI handler. |

### Miscellaneous

| Syntax | Purpose |
|---|---|
| `REM comment` | Program comment; rest of line is ignored by interpreter. |
| `DEF FN name(var)=expr` | Define a single-line user function. |
| `WAIT addr,mask[,xor]` | Busy-wait until `(PEEK(addr) XOR xor) AND mask` is non-zero. |
| `NULL n` | Set the number of null (zero) bytes sent after each carriage return. |

## A.2 Function Reference

Functions return a value and may be used within any expression. String functions are marked with a `$` suffix.

### Numeric Functions

| Function | Returns |
|---|---|
| `SGN(n)` | Sign of $n$: $-1$, 0, or 1. |
| `INT(n)` | Truncate toward zero to integer. |
| `ABS(n)` | Absolute value. |
| `SQR(n)` | Square root. |
| `RND(n)` | Pseudo-random number in $[0, 1)$; $n$ seeds or advances the sequence. |
| `LOG(n)` | Natural logarithm ($\ln n$). |
| `EXP(n)` | $e$ raised to the power $n$. |
| `SIN(n)` | Sine of $n$ radians. |
| `COS(n)` | Cosine of $n$ radians. |
| `TAN(n)` | Tangent of $n$ radians. |
| `ATN(n)` | Arctangent of $n$, result in radians. |
| `PI` | Constant $\pi \approx 3.14159\ldots$ |
| `TWOPI` | Constant $2\pi \approx 6.28318\ldots$ |
| `MAX(a,b)` | The larger of two numeric values. |
| `MIN(a,b)` | The smaller of two numeric values. |
| `FRE(x)` | Free BASIC program memory in bytes ($x$ is ignored). |
| `POS(x)` | Current text cursor column position ($x$ is ignored). |
| `USR(x)` | Call user machine-code routine; pass $x$ in FAC, return value in FAC. |

### String Functions

| Function | Returns |
|---|---|
| `LEN(s$)` | Length of string in characters. |
| `ASC(s$)` | ASCII code of the first character. |
| `CHR$(n)` | Single-character string for ASCII code $n$. |
| `STR$(n)` | Numeric value converted to a string. |

| Function | Returns |
|---|---|
| `VAL(s$)` | Numeric value parsed from the leading digits of a string. |
| `LEFT$(s$,n)` | First $n$ characters of string. |
| `RIGHT$(s$,n)` | Last $n$ characters of string. |
| `MID$(s$,start[,len])` | Substring starting at *start* (1-based), optional length. |
| `UCASE$(s$)` | String converted to uppercase. |
| `LCASE$(s$)` | String converted to lowercase. |
| `HEX$(n)` | Hexadecimal string representation of integer $n$. |
| `BIN$(n)` | Binary string representation of integer $n$. |
| `SADD(s$)` | Address of the string's data in the string pool. |
| `VARPTR(var)` | Address of a numeric or string variable in memory. |

### Sprite and Graphics Functions

| Function | Returns |
|---|---|
| `SPRITEX(n)` | X position of sprite $n$ (currently always returns 0). |
| `SPRITEY(n)` | Y position of sprite $n$ (currently always returns 0). |
| `COLLISION(n)` | Sprite-to-sprite collision bitmask for sprite $n$. |
| `BUMPED(n)` | Sprite-to-background collision bitmask for sprite $n$. |

`SPRITEX()` and `SPRITEY()` are present as ROM tokens. In v1.0 the ROM handlers return 0; position read-back is not yet implemented.

### Music Functions

| Function | Returns |
|---|---|
| `PLAYING` | 1 if music is currently playing, 0 if stopped. |
| `MNOTE(voice)` | Current MIDI note number on *voice* (1–3), or 0 if silent. |

## A.3 Token Index

This index is derived directly from the `TK_*` and `XTK_*` symbol definitions in `ehbasic/basic.asm`. Single-byte tokens begin at `$80`. Extended two-byte tokens use a `$FF` escape prefix followed by the `XTK_*` byte.

### Primary Statement Tokens (TK_END through TK_NMI)

```
END, FOR, NEXT, DATA, INPUT, DIM, READ, LET, DEC, GOTO, RUN, IF, RESTORE, GOSUB,
RETIRQ, RETNMI, RETURN, REM, STOP, ON, NULL, INC, WAIT, LOAD, SAVE, DEF, POKE,
DOKE, CALL, DO, LOOP, PRINT, CONT, LIST, CLEAR, NEW, WIDTH, GET, SWAP, BITSET,
BITCLR, IRQ, NMI
```

### Graphics and Sound Statement Tokens (TK_CLS through TK_VSYNC)

```
CLS, COLOR, LOCATE, PLOT, UNPLOT, LINE, CIRCLE, RECT, FILL, PAINT, MODE, GCLS,
GCOLOR, SPRITE, SPRITESHAPE, SPRITECOLOR, SPRITEDATA, SOUND, VOLUME, INSTRUMENT,
WAVE (deprecated), VSYNC
```

### Secondary Tokens (TK_TAB through TK_OFF)

```
TAB, ELSE, TO, FN, SPC, THEN, NOT, STEP, UNTIL, WHILE, OFF
```

### Operator Tokens

```
+, -, *, /, ^, AND, EOR, OR, » (RSHIFT), « (LSHIFT), >, =, <
```

### Function Tokens (TK_SGN through TK_BUMPED)

```
SGN, INT, ABS, USR, FRE, POS, SQR, RND, LOG, EXP, COS, SIN, TAN, ATN, PEEK,
DEEK, SADD, LEN, STR$, VAL, ASC, UCASE$, LCASE$, CHR$, HEX$, BIN$, BITTST, MAX,
MIN, PI, TWOPI, VARPTR, LEFT$, RIGHT$, MID$, SPRITEX(), SPRITEY(), COLLISION(),
BUMPED()
```

### Extended Two-Byte Tokens ($FF prefix, XTK_DIR through XTK_MNOTE)

```
DIR, DEL, XMEM, XBANK, XPOKE, XPEEK(), STASH, FETCH, XFREE, XRESET, XALLOC,
XDIR, XDEL, XMAP, XUNMAP, GSAVE, GLOAD, SIDPLAY, SIDSTOP, MUSIC, PLAYING,
MNOTE()
```

Chapter B

# Memory Map

## B.1 Address Space Overview

The e6502 virtual computer presents a flat 64 KB address space to the 6502 CPU. Coprocessor regions (VGC, VSC, FIO, XMC) respond to reads and writes within their assigned windows; all remaining space is RAM except the upper 16 KB ($C000–$FFFF) which is write-protected ROM.

| Address Range | Size | Region |
|---|---|---|
| $0000-$00FF | 256 B | Zero Page |
| $0100-$01FF | 256 B | CPU Stack |
| $0200-$027F | 128 B | System Vectors (IRQ/NMI handlers, BASIC vectors) |
| $0280-$9FFF | 39,680 B | BASIC Program RAM |
| $A000-$A01E | 31 B | Virtual Graphics Controller (VGC) registers and command interface |
| $A100-$A1FF | 256 B | Virtual Sound Controller (VSC) registers |
| $AA00-$B1CF | 2,000 B | Character RAM (80×25 text cells) |
| $B1D0-$B99F | 2,000 B | Color RAM (80×25 text cells) |
| $B9A0-$B9EF | 80 B | File I/O Controller (FIO) registers |
| $BA00-$BA3F | 64 B | Expansion Memory Controller (XMC) registers |
| $BA40-$BA4F | 16 B | Timer Controller registers |
| $BA50-$BA53 | 4 B | Music Status and Voice Note Readback |
| $BC00-$BFFF | 1,024 B | XMC Memory Windows (4 × 256-byte mapped pages) |
| $C000-$FFFF | 16,384 B | ROM (NovaBASIC interpreter) |
| $D400-$D41C | 29 B | SID chip registers (inside ROM range; writes intercepted) |

The address range $A01F–$A0FF and $A200–$A9FF are not claimed by any coprocessor and fall through to the underlying flat RAM. The range $BA54–$BBFF is similarly unallocated RAM. SID registers at $D400–$D41C occupy space within the ROM address range but are intercepted on write by the SID chip emulator.

## B.2 VGC Register Map

The Virtual Graphics Controller occupies $A000–$A01E. Registers $A000–$A00F are the core status and display registers. Registers $A010–$A01E are the command register and its 14 parameter slots; writing to $A010 both stores the command byte and triggers immediate execution.

### Core Registers ($A000–$A00F)

| Address | Name | Access | Description |
|---|---|---|---|
| $A000 | RegMode | R/W | Display mode: 0=text only, 1=graphics over text, 2=text over graphics. |
| $A001 | RegBgCol | R/W | Background color index (0–15). |
| $A002 | RegFgCol | R/W | Default foreground color index (0–15); reset value is 1 (white). |
| $A003 | RegCursorX | R/W | Text cursor column (0–79). |
| $A004 | RegCursorY | R/W | Text cursor row (0–24). |
| $A005 | RegScrollX | R/W | Horizontal scroll offset (used by copper raster effects). |
| $A006 | RegScrollY | R/W | Vertical scroll offset (used by copper raster effects). |
| $A007 | RegBank | R/W | Reserved. |
| $A008 | RegStatus | RO | Frame counter; incremented each video frame. Writes are ignored. |
| $A009 | RegSpriteCount | RO | Count of currently enabled sprites (0–16). Writes are ignored. |
| $A00A | RegCursorEna | R/W | Non-zero enables the cursor blink. |
| $A00B | RegColSt | RO | Sprite-to-sprite collision bitmask; reading clears the register. |
| $A00C | RegColBg | RO | Sprite-to-background collision bitmask; reading clears the register. |
| $A00D | RegBorder | R/W | Border color index (0–15). |
| $A00E | RegCharOut | R/W | Character output port; writing outputs a character to the text screen. |
| $A00F | RegCharIn | R/W | Character input port; reading dequeues the next keypress byte. |

### Command Register and Parameters ($A010–$A01E)

| Address | Name | Access | Description |
|---|---|---|---|
| $A010 | RegCmd | R/W | Command byte; writing triggers immediate command execution. |
| $A011 | RegP0 | R/W | Parameter 0. |
| $A012 | RegP1 | R/W | Parameter 1. |
| $A013 | RegP2 | R/W | Parameter 2. |
| $A014 | RegP3 | R/W | Parameter 3. |
| $A015 | RegP4 | R/W | Parameter 4. |
| $A016 | RegP5 | R/W | Parameter 5. |

| Address | Name | Access | Description |
|---|---|---|---|
| $A017 | RegP6 | R/W | Parameter 6. |
| $A018 | RegP7 | R/W | Parameter 7. |
| $A019 | RegP8 | R/W | Parameter 8. |
| $A01A | RegP9 | R/W | Parameter 9. |
| $A01B | RegP10 | R/W | Parameter 10. |
| $A01C | RegP11 | R/W | Parameter 11. |
| $A01D | RegP12 | R/W | Parameter 12. |
| $A01E | RegP13 | R/W | Parameter 13. |

Multi-byte parameters (coordinates, sprite positions) are packed little-endian across consecutive parameter registers. For example, a 16-bit x-coordinate uses P0 (low byte) and P1 (high byte).

## B.3 VGC Command Codes

All commands are invoked by writing the command byte to RegCmd ($A010). Parameters must be loaded into RegP0–RegP13 before the write.

### Graphics Commands ($01–$09)

| Code | Name | Parameters and behavior |
|---|---|---|
| $01 | CmdPlot | P0/P1 = x (16-bit), P2/P3 = y (16-bit). Set pixel to current draw color. |
| $02 | CmdUnplot | P0/P1 = x (16-bit), P2/P3 = y (16-bit). Clear pixel to 0 (transparent). |
| $03 | CmdLine | P0/P1 = x0, P2/P3 = y0, P4/P5 = x1, P6/P7 = y1. Draw Bresenham line. |
| $04 | CmdCircle | P0/P1 = cx, P2/P3 = cy, P4/P5 = radius. Draw circle outline. |
| $05 | CmdRect | P0/P1 = x0, P2/P3 = y0, P4/P5 = x1, P6/P7 = y1. Draw rectangle outline. |
| $06 | CmdFill | P0/P1 = x0, P2/P3 = y0, P4/P5 = x1, P6/P7 = y1. Draw filled rectangle. |
| $07 | CmdGcls | No parameters. Clear entire graphics bitmap to 0. |
| $08 | CmdGcolor | P0 low nibble = color index (0–15). Set current draw color. |
| $09 | CmdPaint | P0/P1 = x (16-bit), P2/P3 = y (16-bit). Flood-fill from seed point. |

### Sprite Commands ($10–$18)

| Code | Name | Parameters and behavior |
|---|---|---|
| $10 | CmdSprDef | P0 = sprite (0–15), P1 = x pixel (0–15), P2 = y pixel (0–15), P3 = color nibble. Set one pixel in sprite shape. |

| Code | Name | Parameters and behavior |
|---|---|---|
| $11 | CmdSprRow | P0 = sprite (0–15), P1 = row (0–15), P2–P9 = 8 data bytes (two 4-bit pixels per byte). Define one sprite row. |
| $12 | CmdSprClr | P0 = sprite (0–15). Clear all 128 bytes of sprite shape data to 0. |
| $13 | CmdSprCopy | P0 = source sprite (0–15), P1 = destination sprite (0–15). Copy shape data. |
| $14 | CmdSprPos | P0 = sprite (0–15), P1/P2 = x (16-bit), P3/P4 = y (16-bit). Set screen position. |
| $15 | CmdSprEna | P0 = sprite (0–15). Enable sprite; increments RegSpriteCount. |
| $16 | CmdSprDis | P0 = sprite (0–15). Disable sprite; decrements RegSpriteCount. |
| $17 | CmdSprFlip | P0 = sprite (0–15), P1 = flags (0=none, 1=horizontal, 2=vertical, 3=both). |
| $18 | CmdSprPri | P0 = sprite (0–15), P1 = priority (0=behind all, 1=between text/gfx, 2=in front). |

Sprite shape data is stored host-side and is not 6502-addressable. All sprite shape manipulation must go through the command register interface.

### Memory I/O Commands ($19–$1A)

| Code | Name | Parameters and behavior |
|---|---|---|
| $19 | CmdMemRead | P0 = memory space (0–3), P1/P2 = address (16-bit), P4 bit 0 = auto-increment. Read byte from VGC memory; result in P3. |
| $1A | CmdMemWrite | P0 = memory space (0–3), P1/P2 = address (16-bit), P3 = data byte, P4 bit 0 = auto-increment. Write byte to VGC memory. |

Memory spaces: 0=character RAM (2000 B), 1=color RAM (2000 B), 2=graphics bitmap (64000 B), 3=sprite shape RAM (2048 B). Auto-increment advances the address after each read or write.

### Copper Commands ($1B–$1E)

The copper triggers register writes at specific raster positions each frame.

| Code | Name | Parameters and behavior |
|---|---|---|
| $1B | CmdCopperAdd | P0/P1 = X (16-bit), P2 = Y, P3/P4 = register (0–15 or $A000–$A00F), P5 = value. Replaces existing event at same position/register. Max 1024 events. |
| $1C | CmdCopperClear | No parameters. Remove all copper events. |

| Code | Name | Parameters and behavior |
|---|---|---|
| $1D | CmdCopperEnable | No parameters. Start executing copper program each frame. |
| $1E | CmdCopperDisable | No parameters. Stop executing copper program. |

Copper-writable registers: RegMode ($A000), RegBgCol ($A001), RegScrollX ($A005), RegScrollY ($A006).

## B.4 SID Chip Registers

The SID chip occupies $D400–$D41C within the ROM address range. Writes to these addresses are intercepted by the SID emulator; reads return the underlying ROM byte. The register layout matches the original MOS 6581.

### Per-Voice Registers (3 voices, 7 bytes each)

| Offset | Description |
|---|---|
| +0 | Frequency low byte. |
| +1 | Frequency high byte (16-bit SID frequency units). |
| +2 | Pulse width low byte. |
| +3 | Pulse width high byte (12-bit, bits 0–11 only). |
| +4 | Control register: bit 0=gate, bit 4=triangle, bit 5=sawtooth, bit 6=pulse, bit 7=noise. |
| +5 | Attack (bits 7–4) / Decay (bits 3–0). |
| +6 | Sustain (bits 7–4) / Release (bits 3–0). |

Voice 0: $D400–$D406. Voice 1: $D407–$D40D. Voice 2: $D40E–$D414.

### Filter and Volume Registers

| Address | Description |
|---|---|
| $D415 | Filter cutoff low (bits 0–2). |
| $D416 | Filter cutoff high (bits 0–7). |
| $D417 | Resonance (bits 7–4) / Filter route (bits 3–0, one bit per voice + external). |
| $D418 | Volume (bits 3–0) / Filter mode (bit 4=LP, bit 5=BP, bit 6=HP). |

The BASIC commands `INSTRUMENT`, `SOUND`, and `MUSIC` manage SID registers automatically. Direct writes to $D400+ are for advanced use only and may conflict with the music engine.

## B.5 Timer Controller and Music Status

### Timer Controller ($BA40–$BA4F)

The timer controller provides periodic interrupt generation for the SID player. Configuration is handled automatically by `SIDPLAY`.

### Music Status ($BA50–$BA53)

| Address | Access | Description |
|---|---|---|
| $BA50 | RO | Status flags: bit 0 = SFX playing, bit 1 = music playing. |
| $BA51 | RO | Voice 1 current MIDI note (0 = silent). |
| $BA52 | RO | Voice 2 current MIDI note (0 = silent). |
| $BA53 | RO | Voice 3 current MIDI note (0 = silent). |

These registers are read by the `PLAYING` and `MNOTE()` functions.

## B.6 VSC Register Map

The Virtual Sound Controller occupies $A100–$A1FF. Writing to $A100 (`VscCmd`) both stores the command byte and executes it.

### VSC Registers

| Address | Name | Access | Description |
|---|---|---|---|
| $A100 | VscCmd | R/W | Command byte; writing triggers execution. |
| $A101 | VscP0 | R/W | Parameter 0. |
| $A102 | VscP1 | R/W | Parameter 1. |
| $A103 | VscP2 | R/W | Parameter 2. |
| $A104 | VscP3 | R/W | Parameter 3. |
| $A105 | VscP4 | R/W | Parameter 4. |
| $A106 | VscP5 | R/W | Parameter 5. |
| $A107 | VscP6 | R/W | Parameter 6. |
| $A108 | VscP7 | R/W | Parameter 7. |
| $A10E | VscActiveMask | RO | Bitmask of currently playing channels (bits 0–3). |
| $A10F | VscMasterVol | RO | Current master volume (0–15); set via `VscCmdVolume`. |

### VSC Command Codes

| Code | Name | Parameters |
|---|---|---|
| $01 | VscCmdSound | P0 = channel (masked to 0–3), P1/P2 = frequency (16-bit Hz), P3/P4 = duration (16-bit, units of 1/60 s). Frequency is clamped to 16–12000 Hz. `freq<=0` or `dur<=0` stops the channel. |
| $02 | VscCmdVolume | P0 low nibble = master volume (0–15). |
| $03 | VscCmdEnvelope | P0 = channel (masked to 0–3), P1 = attack, P2 = decay, P3 = sustain (low nibble, 0–15 scale), P4 = release. Each time value is scaled to samples internally. |

| Code | Name | Parameters |
|---|---|---|
| $04 | VscCmdWave | P0 = channel (masked to 0–3), P1 = waveform index (P1 mod 5): 0=square, 1=sawtooth, 2=triangle, 3=noise, 4=sine. |

## B.7 FIO Register Map

The File I/O Controller occupies $B9A0–$B9EF. Writing to $B9A0 (FioCmd) triggers the operation. The caller polls $B9A1 (FioStatus) for completion.

### FIO Registers

| Address | Name | Access | Description |
|---|---|---|---|
| $B9A0 | FioCmd | R/W | Command byte; writing triggers the operation. |
| $B9A1 | FioStatus | RO | Result status: 0=idle, 2=ok, 3=error. |
| $B9A2 | FioErrCode | RO | Error detail code (see below). |
| $B9A3 | FioNameLen | R/W | Filename length in bytes (1–63). |
| $B9A4 | FioSrcL | R/W | Source/destination address, low byte. |
| $B9A5 | FioSrcH | R/W | Source/destination address, high byte. |
| $B9A6 | FioEndL | R/W | End address, low byte (used by SAVE to determine program extent). |
| $B9A7 | FioEndH | R/W | End address, high byte. |
| $B9A8 | FioSizeL | RO | Loaded data size, low byte (written by host after LOAD or DIR read). |
| $B9A9 | FioSizeH | RO | Loaded data size, high byte. |
| $B9B0-$B9EF | FioName | R/W | Filename buffer (64 bytes ASCII, not null-terminated). |

### FIO Command Codes

| Code | Name | Behavior |
|---|---|---|
| $01 | FioCmdSave | Save bytes from FioSrcL/H to FioEndL/H (exclusive) to disk; prepends a 2-byte load-address. |
| $02 | FioCmdLoad | Load file into RAM at FioSrcL/H; skips the 2-byte load-address prefix; sets FioSizeL/H. |
| $03 | FioCmdDirOpen | Open the program directory; populates FioName and FioSizeL/H with the first entry. |
| $04 | FioCmdDirRead | Advance to the next directory entry; populates FioName and FioSizeL/H. |
| $05 | FioCmdDelete | Delete the named program from disk. |
| $06 | FioCmdGSave | Save VGC memory space to a .gfx file. FioGSpace=space, FioGAddrL/H=offset, FioGLenL/H=length. |

| Code | Name | Behavior |
|---|---|---|
| $07 | FioCmdGLoad | Load .gfx file into VGC memory space. FioGSpace=space, FioGAddrL/H=offset, FioGLenL/H=max length. |
| $08 | FioCmdSidPlay | Load and play a .sid file. FioSrcL=song number (1-based). |
| $09 | FioCmdSidStop | Stop SID file playback. |
| $0A | FioCmdInstrument | Define instrument preset. FioSrcL=id, FioSrcH=waveform, FioEndL=A, FioEndH=D, FioSizeL=S, FioSizeH=R. |
| $0B | FioCmdSound | Play SFX. FioSrcL=MIDI note, FioSrcH=duration (frames), FioEndL=instrument ID. |
| $0C | FioCmdVolume | Set SID master volume. FioSrcL=level (0–15). |
| $0D | FioCmdMSeq | Load MML sequence. FioSrcL=voice (1–3), FioEndL/H=string pointer, FioNameLen=string length. |
| $0E | FioCmdMPlay | Start music playback. |
| $0F | FioCmdMStop | Stop music playback. |
| $10 | FioCmdMTempo | Set tempo. FioSrcL/H=BPM (16-bit). |
| $11 | FioCmdMLoop | Set loop. FioSrcL=0 (off) or 1 (on). |

### FIO Status Codes

| Value | Name | Meaning |
|---|---|---|
| $00 | FioStatusIdle | No operation in progress. |
| $02 | FioStatusOk | Operation completed successfully. |
| $03 | FioStatusError | Operation failed; see FioErrCode. |

### FIO Error Codes

| Value | Name | Meaning |
|---|---|---|
| $00 | FioErrNone | No error. |
| $01 | FioErrNotFound | File not found on disk. |
| $02 | FioErrIo | Host I/O error (invalid name, end address ≤ start, OS exception). |
| $03 | FioErrEndOfDir | No more directory entries (returned for DirOpen on empty dir or after last entry). |

## B.8 XMC Register Map

The Expansion Memory Controller occupies $BA00–$BA3F. Writing to $BA00 (XmcCmd) triggers the operation. Memory windows ($BC00–$BFFF) provide direct CPU-bus access to mapped XRAM pages.

### XMC Registers

| Address | Name | Access | Description |
|---|---|---|---|
| $BA00 | XmcCmd | R/W | Command byte; writing triggers execution. |
| $BA01 | XmcStatus | RO | Result status: 0=idle, 2=ok, 3=error. |
| $BA02 | XmcErrCode | RO | Error detail code (see below). |
| $BA03 | XmcCfg | R/W | Reserved. |
| $BA04 | XmcAddrL | R/W | XRAM address, low byte. |
| $BA05 | XmcAddrM | R/W | XRAM address, middle byte. |
| $BA06 | XmcAddrH | R/W | XRAM address, high byte. |
| $BA07 | XmcRamL | R/W | CPU RAM address, low byte. |
| $BA08 | XmcRamH | R/W | CPU RAM address, high byte. |
| $BA09 | XmcLenL | R/W | Transfer length, low byte. |
| $BA0A | XmcLenH | R/W | Transfer length, high byte. |
| $BA0B | XmcData | R/W | Byte data port (used by GetByte/PutByte). |
| $BA0C | XmcBank | R/W | Default 64 KB bank selector. |
| $BA0D | XmcBanks | RO | Total number of 64 KB banks available (read-only). |
| $BA0E | XmcPagesUsedL | RO | Used 256-byte pages, low byte. |
| $BA0F | XmcPagesUsedH | RO | Used 256-byte pages, high byte. |
| $BA10 | XmcPagesFreeL | RO | Free 256-byte pages, low byte. |
| $BA11 | XmcPagesFreeH | RO | Free 256-byte pages, high byte. |
| $BA12 | XmcNameLen | R/W | Name length for named block operations (1–28). |
| $BA13 | XmcHandle | RO | Block handle returned by Alloc/NStash/DirRead. |
| $BA14 | XmcDirCountL | RO | Count of named blocks, low byte. |
| $BA15 | XmcDirCountH | RO | Count of named blocks, high byte. |
| $BA16 | XmcWinCtl | R/W | Window enable bitmask (bit 0=window 0, bit 1=window 1, etc.). |
| $BA18 | XmcWin0AL | R/W | Window 0 mapped XRAM base address, low byte. |
| $BA19 | XmcWin0AM | R/W | Window 0 mapped XRAM base address, middle byte. |
| $BA1A | XmcWin0AH | R/W | Window 0 mapped XRAM base address, high byte. |
| $BA1B | XmcWin1AL | R/W | Window 1 mapped XRAM base address, low byte. |
| $BA1C | XmcWin1AM | R/W | Window 1 mapped XRAM base address, middle byte. |
| $BA1D | XmcWin1AH | R/W | Window 1 mapped XRAM base address, high byte. |
| $BA1E | XmcWin2AL | R/W | Window 2 mapped XRAM base address, low byte. |
| $BA1F | XmcWin2AM | R/W | Window 2 mapped XRAM base address, middle byte. |
| $BA20 | XmcWin2AH | R/W | Window 2 mapped XRAM base address, high byte. |
| $BA21 | XmcWin3AL | R/W | Window 3 mapped XRAM base address, low byte. |

| Address | Name | Access | Description |
|---|---|---|---|
| $BA22 | XmcWin3AM | R/W | Window 3 mapped XRAM base address, middle byte. |
| $BA23 | XmcWin3AH | R/W | Window 3 mapped XRAM base address, high byte. |
| $BA24-$BA3F | XmcName | R/W | ASCII name buffer (28 bytes, not null-terminated). |

### XMC Command Codes

| Code | Name | Behavior |
|---|---|---|
| $01 | XmcCmdGetByte | Read byte at XmcAddrL/M/H into XmcData. |
| $02 | XmcCmdPutByte | Write XmcData to XmcAddrL/M/H; marks page used. |
| $03 | XmcCmdStash | Copy XmcLenL/H bytes from CPU RAM at XmcRamL/H to XRAM at XmcAddrL/M/H. len=0 is a no-op success. |
| $04 | XmcCmdFetch | Copy XmcLenL/H bytes from XRAM at XmcAddrL/M/H to CPU RAM at XmcRamL/H. len=0 is a no-op success. |
| $05 | XmcCmdFill | Fill XmcLenL/H bytes in XRAM starting at XmcAddrL/M/H with XmcData. |
| $07 | XmcCmdStats | Refresh the PagesUsed/PagesFree/DirCount read-only registers. |
| $08 | XmcCmdResetUsage | Clear all usage tracking, block records, and named-block metadata (destructive). |
| $09 | XmcCmdRelease | Mark XRAM range (XmcAddrL/M/H, XmcLenL/H) as free; removes any overlapping block records. |
| $0A | XmcCmdAlloc | Allocate XmcLenL/H bytes; sets XmcAddrL/M/H, XmcHandle, and XmcBank. |
| $0B | XmcCmdNStash | Named stash: create or update named block from CPU RAM; name read from XmcName/XmcNameLen. |
| $0C | XmcCmdNFetch | Named fetch: copy named block to CPU RAM at XmcRamL/H; len=0 fetches full block. |
| $0D | XmcCmdNDelete | Delete named block by name. |
| $0E | XmcCmdNDirOpen | Open named-block directory; emits first entry to registers. |
| $0F | XmcCmdNDirRead | Advance to the next named-block directory entry. |

### XMC Status Codes

| Value | Name | Meaning |
|---|---|---|
| $00 | XmcStatusIdle | No operation in progress. |
| $02 | XmcStatusOk | Operation completed successfully. |
| $03 | XmcStatusError | Operation failed; see XmcErrCode. |

### XMC Error Codes

| Value | Name | Meaning |
|---|---|---|
| $00 | XmcErrNone | No error. |
| $01 | XmcErrRange | XRAM address or length out of bounds. |
| $02 | XmcErrBadArgs | Invalid arguments (e.g., len<=0 for Alloc, unknown command). |
| $03 | XmcErrNotFound | Named block not found. |
| $04 | XmcErrNoSpace | No contiguous free pages of the required size, or handle pool exhausted. |
| $05 | XmcErrName | Name length is 0 or exceeds 28, or name is blank after trimming. |
| $06 | XmcErrEndOfDir | No more named-block directory entries. |

## B.9 System Vectors

The address range $0200–$027F is the system vector table. Each entry is a 16-bit little-endian pointer initialized from ROM at cold start. BASIC uses the lower portion; the upper portion is reserved for future use.

| Address | Purpose |
|---|---|
| $0200-$0201 | IRQ handler vector (2 bytes, little-endian). Set by the `IRQ` statement. |
| $0202-$0203 | NMI handler vector (2 bytes, little-endian). Set by the `NMI` statement. |
| $0204-$020D | Reserved BASIC internal vectors (warm-start, error, output, input hooks). |
| $020E-$027F | Reserved for future system use. |

The exact layout of $0204–$020D is inherited from EhBASIC 2.22p5 and tracks the standard warm-start, error, and I/O indirection vectors. Refer to `ehbasic/basic.asm` for the definitive symbol assignments.

Chapter C

# Limits, Errors, and Edge Cases

This appendix documents all known numeric limits, argument validation rules, and edge-case behaviors derived from a direct audit of the ROM source and the Avalonia hardware controller implementations. Where ROM behavior and host behavior differ, both are noted.

## C.1 Numeric Argument Conversion Rules

Most command arguments are passed through one of two shared ROM helpers before reaching the hardware layer. Understanding their constraints prevents unexpected function-call errors.

| Helper | Behavior |
|---|---|
| `LAB_GTBY` | Converts the FAC (floating-point accumulator) to an unsigned byte. Accepts values 0–255. Any value outside this range, or any negative value, raises a function-call error before the command reaches the hardware. |
| `LAB_GTWRD` | Converts the FAC to an unsigned 16-bit integer. Accepts values 0–65535. Negative values or values above 65535 raise a function-call error. |

**Warning**

Commands that accept addresses (`POKE`, `DOKE`, `CALL`, `WAIT`, `STASH`, `FETCH`...) route through `LAB_GTWRD`. Passing a value such as `-1` will raise an error rather than wrapping to `$FFFF`.

## C.2 File Command Limits

| Topic | Behavior |
|---|---|
| Filename length | The ROM filename parser accepts 1–63 characters. A length of 0 or greater than 63 causes the FIO controller to return an I/O error (the `ReadFilename` guard in `FileIoController.cs`). |
| Allowed filename characters | The host implementation enforces the pattern `[A-Za-z0-9_.\-]+`. Any character outside this set causes `ReadFilename` to return `null` and the operation to fail with `FioErrIo`. |

| Topic | Behavior |
|---|---|
| `.bas` extension | If the filename does not already end in `.bas` (case-insensitive), the host appends it automatically before forming the filesystem path. |
| Missing file | `LOAD "name"` and `DEL "name"` on a non-existent file set `FioStatus=$03` and `FioErrCode=$01` (`FioErrNotFound`). The ROM interprets this as "File not found". |
| I/O fault | Any OS-level exception during `SAVE` / `LOAD` / `DEL` sets `FioStatus=$03` and `FioErrCode=$02` (`FioErrIo`). The ROM surfaces this as "I/O Error". |
| `SAVE` end address | If `FioEndH:FioEndL` $\leq$ `FioSrcH:FioSrcL`, the save is rejected immediately with `FioErrIo` before the file is opened. |
| `DIR` on empty catalog | `DirOpen` sets `FioStatus=$03/FioErrCode=$03` (`FioErrEndOfDir`) when no `.bas` files exist. The ROM `DIR` handler treats this as a silent empty listing. |
| `DIR` after last entry | Each `DirRead` beyond the final file sets `FioErrEndOfDir`; the ROM stops iterating. |

## C.3 Graphics and Sprite Edge Cases

| Command / Feature | Limit and edge behavior |
|---|---|
| `GCOLOR c` | Only the low nibble of $c$ is used (`c & 0x0F`). Values 0–15 are valid; values above 15 wrap silently into the 0–15 range. |
| Color 0 on the graphics layer | Color index 0 means transparent on the graphics bitmap. `UNPLOT x,y` explicitly sets a pixel to 0 to restore transparency. |
| `PLOT` / `UNPLOT` / `PAINT` bounds | The ROM dispatches coordinates without host-side pre-clipping. The host `BlockGraphics` implementation performs pixel-level bounds checks; out-of-range coordinates are silently ignored. |
| `LINE` / `RECT` / `CIRCLE` / `FILL` bounds | Drawing operations are clipped by the host renderer (`BlockGraphics.cs`). Portions of the shape outside the 320×200 pixel area are dropped; no error is raised. |
| `FILL` rectangle | Coordinates are clamped to the screen boundary before drawing; the swap of x0/x1 or y0/y1 to ensure a positive rectangle is handled by the host. |
| `SPRITE n,...` invalid index | Sprite index $n$ must be 0–15. The VGC command handlers check `n >= MaxSprites` and return immediately without error; no ROM-level error is raised. |
| `SPRITEDATA n,row,...` | Row must be 0–15. The `CmdSprRow` handler checks both sprite index and row; an invalid row causes the command to be silently ignored. |

| Command / Feature | Limit and edge behavior |
|---|---|
| `SPRITESHAPE` / `SPRITECOLOR` | These are tokenised and recognised by the ROM parser. However, the ROM handlers for both commands currently perform no hardware action. They exist for source compatibility; no VGC command is issued. |
| `SPRITEX(n)` / `SPRITEY(n)` | The ROM functions return 0 in v1.0. Sprite position read-back from the VGC is not yet implemented. |
| `COLLISION(n)` / `BUMPED(n)` | The VGC updates `RegColSt` and `RegColBg` each frame; the ROM reads the appropriate register and clears it on read. A given bit is set if sprite $n$ participated in a collision that frame. |
| Default sprite priority | On reset all sprites default to priority 2 (in front of everything). This matches the `SpritePriInFront` constant. |
| `CmdSprFlip` flags | Only bits 0–1 of the flags byte are used (`flags & 0x03`): bit 0 = horizontal flip, bit 1 = vertical flip. |
| `CmdSprPri` clamping | Priority values above 2 are clamped to 2 (`Math.Min(value, 2)`). Values 0, 1, and 2 are the only meaningful levels. |

## C.4 Sound and Music Limits

| Command / Feature | Limit and edge behavior |
|---|---|
| `SOUND note,dur[,inst]` | *note* is a MIDI note number (0–127 byte range). *dur* is duration in 1/60-second frames (0–255 byte range). *inst* is instrument slot (0–15, default 0). If *note* or *dur* is 0, the sound is stopped. |
| Master volume | Only the low nibble of the volume byte is used (`level & 0x0F`). The default master volume on power-on is 12. |
| `INSTRUMENT` parameters | All six parameters are bytes (0–255). Waveform should be one of $10, $20, $40, $80. ADSR values 0–15 are meaningful; higher values use the low nibble only for sustain. |
| `INSTRUMENT` slots | 16 slots (0–15). Slot 0 is pre-initialized at boot. All other slots start as copies of slot 0. |
| `MUSIC voice,...` | Voice must be 1–3. The MML string is read from CPU memory via pointer; maximum practical length limited by available RAM. |
| `MUSIC TEMPO` | BPM is a 16-bit value (0–65535). Default is 120. |
| `MUSIC PRIORITY` | 1–3 voice numbers. Controls which voice is stolen first for SFX. |
| MML pulse width | Range 0–4095. Default 2048. PWM sweep step: ±32 per frame. |
| MML filter cutoff | Range 0–2047. Resonance 0–15. Filter sweep step: ±8 per frame. |
| MML vibrato | Depth is any positive integer; 0 = off. Oscillates at ∼2.9 Hz. |

| Command / Feature | Limit and edge behavior |
|---|---|
| MML loops | Non-nesting. Maximum practical depth limited by string expansion. |
| Copper events | Maximum 1024 events per program. Events at duplicate position/register replace existing values. |
| `WAVE` | Deprecated. Raises a syntax error. Use `INSTRUMENT` instead. |
| `SIDPLAY` | Loads `.sid` files from `~/e6502-programs`. Song number is 1-based (default 1). |

## C.5 XRAM Limits and Failure Modes

| Command / Feature | Limit and edge behavior |
|---|---|
| `XBANK n` | The ROM verifies that $n$ is less than the value stored at `XmcBanks` ($BA0D). An out-of-range bank number triggers a function-call error in the ROM before any XMC command is issued. |
| Window number (`XMAP` / `XUNMAP`) | Window must be 0–3. The ROM validates this; an invalid window triggers a function-call error. |
| Window address space | The four CPU-visible windows occupy $BC00–$BFFF (4 × 256 bytes). Window 0 maps to $BC00, window 1 to $BD00, window 2 to $BE00, window 3 to $BF00. |
| Unmapped window reads/writes | If a window is not enabled in `XmcWinCtl`, the XMC does not own that address and the read/write falls through to flat RAM. No error is returned. |
| Named block name length | The ROM enforces a 1–28 byte name (`XmcNameLen` is capped at 28 by the name buffer size: $BA24–$BA3F = 28 bytes). The host trims whitespace; a blank name after trimming is rejected with `XmcErrName`. |
| Named block name case | Name lookup is case-insensitive in the host (`StringComparer.OrdinalIgnoreCase`). Storing "`SPRITE`" and retrieving "`sprite`" will succeed. |
| `XALLOC len` with `len<=0` | The ROM passes zero through `LAB_GTWRD`, which itself rejects negative values. The XMC command handler rejects `len<=0` with `XmcErrBadArgs`. |
| `XALLOC` with no free space | If no contiguous run of the required pages exists, or the handle pool (1–255) is exhausted, the command fails with `XmcErrNoSpace`. |
| `STASH` / `FETCH` (raw) with `len=0` | The XMC host treats a zero-length raw transfer as a no-op and returns `XmcStatusOk`. No data is moved and no pages are marked. |
| `FETCH "name",ram` (named fetch) | The ROM sends `XmcLenL/H = 0` for the named-fetch command. The host interprets `requested=0` as "fetch the entire stored block": `len = (requested <= 0) ? block.Length : min(requested, block.Length)`. |

| Command / Feature | Limit and edge behavior |
|---|---|
| `STASH "name",ram,len` over existing block | If the named block already exists and the new length fits in the allocated pages, only `block.Length` is updated (no reallocation). If it does not fit, the old block is freed and a new allocation is attempted. |
| `XRESET` | Clears the `_usedPages` array, resets `_usedPageCount` to 0, and removes all block and name records. The raw XRAM byte array is *not* zeroed; data remains but is inaccessible through the allocation system. |
| `XFREE off,len` | Frees all usage-tracking pages in the given range and removes any tracked blocks (named or unnamed) whose page range overlaps with the freed region. |
| `XPOKE` / `XPEEK` bank offset | The ROM constructs the XRAM address as `bank * 65536 + offset`. The host validates the resulting 24-bit address against the total XRAM size; out-of-range addresses return `XmcErrRange`. |
| RAM range validation for STASH/FETCH | The host prevents writes to ROM space: any `FETCH` operation whose destination range would extend into $C000 or above is rejected with `XmcErrRange`. Reads (`STASH`) may source from ROM addresses, allowing code capture. |

## C.6 Status and Error Code Quick Reference

### File I/O Controller (FIO) Codes

| Value | Symbol | Meaning |
|---|---|---|
| *Status codes ($B9A1)* | | |
| $00 | `FioStatusIdle` | No operation has been issued since last reset. |
| $02 | `FioStatusOk` | Last operation succeeded. |
| $03 | `FioStatusError` | Last operation failed; check error code. |
| *Error codes ($B9A2)* | | |
| $00 | `FioErrNone` | No error. |
| $01 | `FioErrNotFound` | File not found on disk. |
| $02 | `FioErrIo` | Host I/O error (bad name, OS exception, end address ≤ start). |
| $03 | `FioErrEndOfDir` | Directory enumeration exhausted. |

### Expansion Memory Controller (XMC) Codes

| Value | Symbol | Meaning |
|---|---|---|
| *Status codes ($BA01)* | | |
| $00 | `XmcStatusIdle` | No operation in progress. |
| $02 | `XmcStatusOk` | Last operation succeeded. |

| Value | Symbol | Meaning |
|---|---|---|
| $03 | XmcStatusError | Last operation failed; check error code. |
| *Error codes ($BA02)* | | |
| $00 | XmcErrNone | No error. |
| $01 | XmcErrRange | XRAM address or transfer endpoint out of XRAM bounds, or FETCH would write into ROM. |
| $02 | XmcErrBadArgs | Invalid argument (len<=0 for Alloc, unknown command byte). |
| $03 | XmcErrNotFound | Named block not found in directory. |
| $04 | XmcErrNoSpace | No contiguous free pages available, or handle pool (1–255) exhausted. |
| $05 | XmcErrName | Name length 0 or >28, or name is blank after trimming. |
| $06 | XmcErrEndOfDir | No more named-block directory entries. |

The BASIC runtime maps both FIO and XMC error returns to one of three user-visible messages: "File not found", "I/O Error", or a function-call error. For low-level programs that POKE the controller registers directly, use the tables above to interpret the raw status and error bytes.