NOVABASIC
User Guide v1.0

For the e6502 Virtual Computer

Derived from EhBASIC 2.22p5 | 2026 Edition

NovaBASIC v1.0 User Guide

For the e6502 Virtual Computer — 2026 Edition

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

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.

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** $\longrightarrow$ **LIST** $\longrightarrow$ **RUN** $\longrightarrow$ **FIX** $\longrightarrow$ **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.

```
1  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.

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.

```
1  10 SCORE = 0
2  20 NAME$ = "PLAYER ONE"
3  30 LIVES = 3
4  40 PRINT NAME$; " HAS "; LIVES; " LIVES"
5  50 SCORE = SCORE + 100
6  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)`.

```
1  10 DIM A(10)
2  20 FOR I=0 TO 10
3  30   A(I) = I * I
4  40 NEXT I
5  50 FOR I=0 TO 10
6  60   PRINT "A(";I;") = ";A(I)
7  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.

```
1  10 REM read and print two bytes at $0300
2  20 PRINT PEEK(768); PEEK(769)
3  30 REM write a counter value
4  40 POKE 768, 42
5  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.

```
1  10 DIM A(5)
2  20 FOR I=1 TO 5
3  30   A(I) = INT(RND(1) * 100)
4  40 NEXT I
5  50 FOR I=1 TO 5
6  60   PRINT "A(";I;") = ";A(I)
7  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.

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`:

$$\text{SPRITEDATA n, row, b1, b2, b3, b4, b5, b6, b7, b8}$$

- `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 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.

4.7 Try It Now

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

```
1  10 MODE 1 : GCLS : GCOLOR 10
2  20 RECT 10, 10, 309, 189
3  30 GCOLOR 14 : CIRCLE 160, 100, 50
4  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.

Sound and Music

> *"Music is the arithmetic of sounds as optics is the geometry of light."*
>
> — Claude Debussy

NovaBASIC includes a four-voice software synthesizer with independently configurable waveforms, ADSR volume envelopes, and frequency control per channel. Audio renders in real time at 44100 Hz through OpenAL. This chapter explains every command and shows practical patterns for playing notes, building textures, and composing multi-voice music.

5.1 The Sound System

The synthesizer exposes four identical voices on channels 0 through 3. Each channel maintains its own:

- **Waveform** — square, sawtooth, triangle, noise, or sine (set with `WAVE`).

- **ADSR envelope** — attack, decay, sustain, release shaping (set with `ENVELOPE`).

- **Frequency and duration** — triggered with `SOUND` .

A single `VOLUME` command sets the global master output level for all channels. The typical initialization sequence at the top of a program is:

```
10 VOLUME 12
20 WAVE 0, 0
30 ENVELOPE 0, 2, 4, 10, 4
```

Line 10 sets master volume to 12 (out of 15). Line 20 selects a square wave on channel 0. Line 30 sets a fast attack, moderate decay, high sustain, and moderate release on channel 0. After this setup, `SOUND` commands on channel 0 will play with that voice character until any of these settings are changed.

5.2 Playing Notes

SOUND syntax

<div align="center">

`SOUND channel, frequency, duration`

</div>

- `channel` — voice index 0–3. The value is masked with `& 0x03` so larger values wrap back into range silently.

- frequency — pitch in Hertz, clamped to 16–12000 Hz.

- duration — note length in 1/60-second ticks. A value of 60 plays for one second; 30 plays for a half second; 8 plays for about 133 milliseconds.

If frequency is zero or negative, or duration is zero or negative, the channel is stopped immediately.

Useful note frequencies

Note	Frequency (Hz)	Approx. duration for a beat at 120 bpm
Middle C (C4)	262	30 ticks (half second)
D4	294	30 ticks
E4	330	30 ticks
F4	349	30 ticks
G4	392	30 ticks
A4	440	30 ticks
B4	494	30 ticks
C5	524	30 ticks

A simple melody

The following program plays a rising C major scale on channel 0 using a triangle wave:

```
10 VOLUME 12 : WAVE 0, 2
20 ENVELOPE 0, 1, 4, 12, 4
30 DATA 262, 294, 330, 349, 392, 440, 494, 524
40 FOR N = 1 TO 8
50   READ F
60   SOUND 0, F, 24
70   FOR I = 1 TO 24 : VSYNC : NEXT I
80 NEXT N
```

Each note plays for 24 ticks ($\approx$ 400 ms) and the VSYNC loop holds the program for exactly the same duration before triggering the next note, keeping the melody in rhythm.

> The duration parameter in SOUND controls how long the synthesizer sustains that note in its internal queue; it does not pause program execution. Use a VSYNC loop after each SOUND call to create the gap between notes.

5.3 Waveforms

WAVE channel, waveform selects the oscillator type for a channel. The waveform index is taken modulo 5, so values 5, 10, etc. wrap back to 0.

Index	Name	Character
0	Square	Bold, hollow, retro chiptune feel. Classic 8-bit game sound.
1	Sawtooth	Buzzy and harmonically rich. Good for brass-like lead sounds and bass lines.
2	Triangle	Soft and mellow, closer to a flute or a gentle bell. Lower-intensity background pads.
3	Noise	Unpitched percussive texture. Use for drum hits, explosions, static, and ambient rumble.
4	Sine	Pure, smooth, and clean. No overtones; ideal for test tones and ethereal pads.

Waveform changes take effect on the next `SOUND` trigger; they do not interrupt a note that is already playing on that channel.

5.4 ADSR Envelopes

A volume envelope shapes how a note's amplitude changes over time. NovaBASIC implements a standard four-stage ADSR envelope:

```
ENVELOPE channel, attack, decay, sustain, release
```

- **Attack** — ramp from silence (0.0) to full amplitude (1.0). Byte value; converted to samples at 44100 Hz / 240 per unit. A value of 0 gives an instantaneous start; larger values give a slower fade-in.

- **Decay** — ramp from full amplitude down to the sustain level. Same timing conversion as attack.

- **Sustain** — the amplitude held while the note duration runs. Only the low nibble is used (0–15), mapped linearly to 0.0–1.0. A value of 15 sustains at full amplitude after the decay phase; a value of 0 gives a purely percussive shape.

- **Release** — ramp from the sustain level back to silence after the note duration expires. Same timing conversion as attack and decay.

Sustain is a *level* (0–15), not a duration. Attack, decay, and release are *timing* values (byte duration counts). This is the standard ADSR model: sustain holds until the note ends, then release fades from there.

Envelope comparison

A short percussive hit vs. a slow rising pad:

```
10 VOLUME 12
20 REM PERCUSSIVE: FAST ATTACK, FAST DECAY, NO SUSTAIN, MEDIUM
   RELEASE
30 WAVE 0, 3
40 ENVELOPE 0, 0, 3, 0, 8
50 SOUND 0, 200, 10
60 FOR I = 1 TO 20 : VSYNC : NEXT I
70 REM SLOW PAD: SLOW ATTACK, MEDIUM DECAY, HIGH SUSTAIN, SLOW
   RELEASE
```

```
8   80 WAVE 1, 4
9   90 ENVELOPE 1, 20, 8, 12, 20
10  100 SOUND 1, 440, 60
11  110 FOR I = 1 TO 90 : VSYNC : NEXT I
```

Channel 0 uses noise (`WAVE 0,3`) with envelope (0, 3, 0, 8) for a tight drum-like thud:
instant attack, quick decay to zero, short release. Channel 1 uses sine (`WAVE 1,4`) with
envelope (20, 8, 12, 20) for a slow fade-in, a gentle decay to a sustained level of 12/15, and a
long fade-out.

5.5 Multi-Voice Music

Because each channel is independent, you can trigger notes on multiple channels simultaneously.
Issue consecutive `SOUND` commands on different channels before calling `VSYNC` ; both will
start rendering within the same frame.

The following example plays a melody on channel 0 and a bass accompaniment on channel 1:

```
1   10 VOLUME 12
2   20 REM MELODY: TRIANGLE, FAST ENVELOPE
3   30 WAVE 0, 2 : ENVELOPE 0, 1, 4, 10, 4
4   40 REM BASS: SAWTOOTH, SLOWER ENVELOPE
5   50 WAVE 1, 1 : ENVELOPE 1, 2, 6, 8, 6
6   60 REM MELODY FREQUENCIES
7   70 DATA 330, 294, 262, 294, 330, 330, 330, 0
8   80 REM BASS FREQUENCIES (HALF SPEED - ONE PER TWO MELODY NOTES)
9   90 DATA 131, 131, 131, 131
10  100 FOR N = 1 TO 8
11  110   READ MF
12  120   IF (N AND 1) = 1 THEN READ BF : SOUND 1, BF, 48
13  130   SOUND 0, MF, 24
14  140   FOR I = 1 TO 24 : VSYNC : NEXT I
15  150 NEXT N
```

Lines 70–90 define the note sequences as `DATA` . The melody plays a new note every 24
ticks. The bass plays one note every 48 ticks by reading a bass frequency only on odd melody
steps (line 120 tests bit 0 of `N`). Both channels render concurrently between `VSYNC` calls.

> Channels do not block each other. A note triggered on channel 0 and a note triggered on
> channel 1 in the same frame play simultaneously with no additional effort.

5.6 Composition Tips

> - Set `VOLUME` and all `ENVELOPE` / `WAVE` values once at program startup, not inside
>   the note loop. Changing them mid-note can cause audible clicks.
>
> - Keep melody, harmony, and percussion on separate channels so each voice can have
>   its own waveform and envelope character.
>
> - Use `VSYNC` loops between note events for tempo control. Each `VSYNC` is exactly
>   1/60 second. At 120 beats per minute (2 beats/second), one beat = 30 VSYNC
>   cycles.

- Store music as `DATA` statements. This separates the note data from the playback engine and makes it easy to edit pitches, durations, and timing without touching the logic.

- Use waveform 3 (noise) on a spare channel for drum and percussion sounds. A short envelope (fast attack, zero sustain) on the noise channel gives convincing hi-hat and snare textures.

5.7 Try It Now

Type and run the following program. It uses a sine wave with a smooth envelope to play an ascending chromatic phrase of eight notes:

```
10 VOLUME 12
20 ENVELOPE 0, 2, 6, 10, 6
30 WAVE 0, 4
40 FOR N = 0 TO 7
50   F = 262 * 2 ^ (N / 12)
60   SOUND 0, INT(F), 8
70   FOR I = 1 TO 8 : VSYNC : NEXT I
80 NEXT N
```

Expected result: eight notes rising by one semitone each, starting at middle C (262 Hz) and ascending chromatically, played with a sine wave and a smooth ADSR envelope. Each note lasts approximately 133 milliseconds (8/60 s).

Experiments to try:

- Change `WAVE 0,4` to `WAVE 0,0` (square) and re-run to hear the same phrase with a retro chiptune character.

- Change the base frequency from 262 to 440 to start the phrase from concert A instead of middle C.

- Increase the loop counter from 7 to 11 for a full chromatic octave.

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.

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
```

Full register-level documentation for every VGC and VSC command — including parameters for drawing, sprites, and sound sequencing — is planned for a future edition of this guide. The current source of truth is `e6502.Avalonia/Hardware/VgcConstants.cs`, which lists every register address and command code with inline comments.

7.7 Future Directions

This chapter will grow as the hardware interfaces stabilize. Planned additions:

- Complete VGC register reference: all drawing commands with full parameter tables.

- VSC register reference: sound and envelope control from assembly.

- Stable calling conventions and register preservation rules for BASIC-to-assembly transitions.

- Worked examples: a full-screen raster effect using XRAM windows and a VSC-driven interrupt music player.

7.8 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.

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 nth line in the list.
ON expr GOSUB l1,l2,...	Call the nth 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

The sound controller provides 4 independent channels (0–3).

Syntax	Purpose
SOUND ch,freq,dur	Play on channel *ch* (0–3); frequency 16–12000 Hz; duration in 1/60 s ticks.
VOLUME level	Set master volume (0–15; only the low nibble is used).
ENVELOPE ch,a,d,s,r	Set ADSR envelope for channel *ch*; sustain uses low nibble (0–15 scale).
WAVE ch,waveform	Set waveform for channel *ch*: 0=square, 1=sawtooth, 2=triangle, 3=noise, 4=sine.

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.

Function	Returns
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.
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.

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, ENVELOPE,
WAVE, 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_XUNMAP`)

```
DIR, DEL, XMEM, XBANK, XPOKE, XPEEK(), STASH, FETCH, XFREE, XRESET, XALLOC,
XDIR, XDEL, XMAP, XUNMAP
```

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
$BC00–$BFFF	1,024 B	XMC Memory Windows (4 × 256-byte mapped pages)
$C000–$FFFF	16,384 B	ROM (NovaBASIC interpreter)

> The address range $A01F–$A0FF and $A200–$A9FF are not claimed by any coprocessor and fall through to the underlying flat RAM. The range $BA40–$BBFF is similarly unallocated RAM.

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 (reserved in v1.0).
$A006	RegScrollY	R/W	Vertical scroll offset (reserved in v1.0).
$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	--	—	Reserved.
$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.
$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.
$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`.

Code	Name	Parameters and behavior
$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.

B.4 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.5 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.

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 $\leq$ start, OS exception).
$03	FioErrEndOfDir	No more directory entries (returned for DirOpen on empty dir or after last entry).

B.6 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.

Address	Name	Access	Description
`$BA12`	`XmcNameLen`	R/W	Name length for named block operations (1–28).
`$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.
`$BA21`	`XmcWin3AL`	R/W	Window 3 mapped XRAM base address, low byte.
`$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.

Code	Name	Behavior
$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.7 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.

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 ≤ 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 Limits and Behavior

Command / Feature	Limit and edge behavior
Channel masking	The channel index is masked to the range 0–3 by the host: `ch = P0 & (ChannelCount - 1)`. Values above 3 wrap into range; no error is raised.
Frequency clamping	Frequency is clamped to 16–12000 Hz by the host (`Math.Clamp(freq, 16, 12000)`). The ROM passes the raw 16-bit value; out-of-range values are silently adjusted rather than rejected.
Stopping a channel	If $freq \leq 0$ or $dur \leq 0$, the channel is immediately deactivated (`Active = false`, `NotePos` and `NoteTotalSamples` reset to 0). This is the correct way to stop a playing note before its duration expires.
Duration units	Duration is in units of 1/60 second (video frames). Internally: `samples = durationTicks * 44100 / 60`.
Master volume	Only the low nibble of the volume byte is used (`P0 & 0x0F`). The default master volume on power-on is 12.
Waveform wrapping	The waveform index is wrapped: `waveform = P1 % 5`. Values 0–4 select square, sawtooth, triangle, noise, and sine respectively.
Sustain level	In `ENVELOPE`, the sustain parameter (P3) uses only the low nibble (0–15 scale), converted to a floating-point level by the host: `SustainLevel = (P3 & 0x0F) / 15.0`.

Command / Feature	Limit and edge behavior
Envelope time parameters	Attack (P1), decay (P2), and release (P4) are converted to sample counts: `samples = max(1, value * 44100 / 240)`. A value of 0 yields 1 sample (instantaneous).

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)`.
`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.

Command / Feature	Limit and edge behavior
`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 $\leq$ 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.
$03	XmcStatusError	Last operation failed; check error code.
Error codes ($BA02)		
$00	XmcErrNone	No error.

Value	Symbol	Meaning
$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.