NovaBASIC User Guide

A Modern BASIC for a Classic CPU

Welcome to NovaBASIC

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

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.

> **Note**
>
> 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`.

Runtime Clock Model

Nova VM runs from a single CPU-cycle clock source. By default the target rate is 12 MHz (12,000,000 cycles/second), and video frame timing is derived from that clock at 60 Hz. This keeps `VSYNC`, raster IRQ behavior, timers, and music sequencing synchronized to one timing model.

You can override runtime behavior with environment variables:

- `NOVA_CPU_HZ`: target CPU rate in cycles/second (default 12,000,000).

- `NOVA_TURBO`: if set to `1`, `true`, `yes`, or `on`, run unthrottled.
- `NOVA_TIMING_LOG`: if set, emit periodic timing telemetry for diagnostics.

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 **320x200 pixel bitmap** with **16 colors** using `PLOT`, `LINE`, `CIRCLE`, `RECT`, `FILL`, and `PAINT`.
- Animate up to **16 hardware sprites**, each 16x16 pixels and multicolor, with per-sprite shape, position, and flip control.
- Play **6-voice synthesized sound** (two SID chips) with ADSR envelopes and four waveforms – triangle, sawtooth, pulse, and noise.
- Save and load programs and data files with `SAVE`, `LOAD`, `DIR`, and `DEL`.
- Access **512KB 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.

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 64KB 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.

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.

Try It

```
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.

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.

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.

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.

Note

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.

Working Efficiently

Good habits established early make everything easier later.

Tip

- **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.

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.

Try It

```
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.

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
```

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
```

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

Control Flow

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
```

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.

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
```

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
```

GOTO

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

Shorthand Commands

DEC and INC

`INC` and `DEC` increment or decrement a variable by 1 without a full assignment statement. They are faster and more readable than `X = X + 1`.

```
10 SCORE = 0
20 INC SCORE : INC SCORE : INC SCORE
30 PRINT "SCORE: "; SCORE
40 DEC SCORE
50 PRINT "AFTER DEC: "; SCORE
```

WAIT

`WAIT addr, mask [, xor]` busy-waits (polls in a tight loop) until a memory location satisfies a bit condition:

`(PEEK(addr) XOR xor) AND mask <> 0`

The *xor* parameter defaults to 0 if omitted. `WAIT` is useful for polling hardware status registers:

```
10 REM Wait for timer IRQ pending (bit 0 of $BA41)
20 WAIT $BA41, 1
```

```
10 REM Wait for a key press (non-zero at $A00F)
20 WAIT $A00F, $FF
```

Warning

`WAIT` blocks the BASIC interpreter completely until the condition is met. If the condition is never satisfied, the program hangs. Use `STOP` (Ctrl+C) to break out.

RESET

`RESET` performs a full hardware reset: it stops both SID chips, halts the music engine, resets the NIC, clears the VGC, and restarts the BASIC interpreter from cold start. All variables and the current program are lost.

```
10 PRINT "RESETTING IN 3..."
20 FOR I=3 TO 1 STEP -1 : PRINT I : NEXT I
30 RESET
```

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

Operator	Type	Description
>=	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 bits by mask at an address
BITCLR	Bit	Clear bits by mask at an address
BITTGL	Bit	Toggle bits by mask at an address
BITTST(a,m)	Bit	Test bits by mask at an address

Built-in Functions

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 3.14159
TWOPI	The constant 2pi 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

String Functions

Function	Description
LEN(s)	*Number of characters in the string*
ASC(s)	*ASCII code of the first character*
CHR(n)	Character whose ASCII code is *n*
STR(n)	Convert number to string

Function	Description
VAL(s)|$Convertstringtonumber$||$LEFT(s,n)$|$Left*n*characters$||$RIGHT(s,n)$|$Right*n*characters$||$MID(s,p,n)$|$*n*charactersstartingatposition*p$||$UCASE(s)$|$Converttouppercase$||$LCASE(s)$|$Converttolowercase$||$HEX$(n)	Hexadecimal string representation of n
BIN$(n)	Binary string representation of n

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 memmap for safe zones.

Cursor Control

The text cursor position is set with `LOCATE x, y` where x is the column (0–79) and y is the row (0–24).

Cursor **visibility** is controlled via the VGC register at $A00A:

```
10 POKE $A00A, 0   : REM hide cursor
20 LOCATE 10, 5    : PRINT "SCORE: 0"
30 POKE $A00A, 1   : REM show cursor
```

Any non-zero value enables the blinking cursor; zero hides it. This is useful in games where you want a clean display without a flashing block.

Style Tips

Tip

- **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.

Putting It Together

Try It

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.

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 320x200 pixel bitmap and a hardware sprite layer. Drawing commands operate on 16 colors; sprites add independently positioned, independently animated 16x16 objects on top of or behind the bitmap and text layers. This chapter covers the full graphics pipeline from mode selection to collision detection.

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.
3	Graphics and sprites only. No text layer is rendered.

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.

> **Note**
>
> 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`.

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.

Command	Description
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.

Tip

`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.

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.

Tip

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.

Sprites

Sprites are hardware-accelerated 16x16 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.

How sprite byte encoding works

The diagram below shows one row of sprite data – 8 bytes producing 16 pixels. Each byte is split into two 4-bit nibbles, each selecting a color index:

In the example above, `$BB` (decimal 187) fills both nibbles with color index `B` (11 = cyan), producing two solid cyan pixels. `$0B` has a transparent left pixel and a cyan right pixel. `$00` leaves both pixels transparent.

The full 16x16 sprite is 16 such rows stacked vertically, each defined by a separate `SPRITEDATA` call. The complete shape occupies 128 bytes (8 bytes x 16 rows):

Each pair of columns shares one byte. The vertical lines mark byte boundaries. Colored cells are non-transparent pixels; gray cells are transparent (color 0).

The following example defines a simple 16x16 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
```

> **Note**
>
> 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.

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"
```

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

Index	Color	Index	Color
6	Blue	14	Light Blue
7	Yellow	15	Light Grey

The background color defaults to 6 (blue), 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).

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

The VGC stores up to 128 independent copper *lists*. Each list is a program of up to 256 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 core registers and the full sprite register block:

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
A040–A0BF	(sprite regs)	All sprite register fields (see below)

COPPER BASIC keyword

The `COPPER` keyword provides direct access to the copper system:

Syntax	Purpose
COPPER ADD x, y, BGCOL, value	Add event: set background color at position.
COPPER ADD x, y, MODE, value	Add event: set display mode at position.
COPPER ADD x, y, SCROLLX, value	Add event: set horizontal scroll at position.
COPPER ADD x, y, SCROLLY, value	Add event: set vertical scroll at position.

Syntax	Purpose
COPPER ADD x, y, SPRX(n), value	Set sprite n X low byte at position.
COPPER ADD x, y, SPRXH(n), value	Set sprite n X high byte at position.
COPPER ADD x, y, SPRY(n), value	Set sprite n Y low byte at position.
COPPER ADD x, y, SPRYH(n), value	Set sprite n Y high byte at position.
COPPER ADD x, y, SPRSHAPE(n), value	Set sprite n shape slot at position.
COPPER ADD x, y, SPRFLAGS(n), value	Set sprite n flags at position.
COPPER ADD x, y, SPRPRI(n), value	Set sprite n priority at position.
COPPER CLEAR	Remove all events from the current target list.
COPPER ON	Enable copper execution each frame.
COPPER OFF	Disable copper execution.
COPPER LIST n	Set target list to n (0–127). Subsequent ADD/CLEAR edit this list.
COPPER LIST END	Reset target list back to the active list.
COPPER USE n	Switch the active list to n at the next vblank.

For sprite register names, n is the sprite index (0–15). Each name maps to the absolute address `$A040 + n*8 + field_offset`. All eight sprite register fields, including `TransColor` (+7), are copper-writable.

If an event at the same position for the same register already exists, the value is replaced. Each list holds up to 256 events.

Multiple lists. The VGC stores 128 copper lists (0–127). By default everything operates on list 0. Use `COPPER LIST n` to direct subsequent `ADD/CLEAR` commands to list n. Use `COPPER USE n` to tell the renderer to switch to list n at the next vertical blank – this avoids tearing because the swap happens atomically between frames. Editing a list and swapping the active pointer are independent, so you can build one list while the renderer displays another (double buffering).

Sprite multiplexing with the copper

The VGC has 16 hardware sprites, but the copper can make them appear as many more by rewriting sprite registers between rows. The technique is called **sprite multiplexing**: after a sprite's row of pixels has been drawn, the copper repositions it further down the screen with a different shape slot, effectively reusing the same hardware sprite for a second (or third) object.

The basic recipe:

1. Set initial sprite positions and shapes for the top of the screen.
2. Use `COPPER ADD` to rewrite `SPRY(n)` and `SPRSHAPE(n)` at scanlines between the rows of sprites, moving the sprite to its next position and swapping its shape.
3. At scanline 0 of the next frame, restore the original values so the top-of-screen sprites reappear.

For example, to display sprite 0 at both Y=20 and Y=120 with different shapes:

```
10 REM Set initial position (top instance)
20 SPRITE 0, 100, 20
30 SPRITESHAPE 0, 0
40 SPRITE 0, ON
50 REM Copper: at scanline 70, reposition to Y=120, shape 1
```

```
60 COPPER ADD 0, 70, SPRY(0), 120
70 COPPER ADD 0, 70, SPRSHAPE(0), 1
80 REM Restore at scanline 0 for next frame
90 COPPER ADD 0, 0, SPRY(0), 20
100 COPPER ADD 0, 0, SPRSHAPE(0), 0
110 COPPER ON
```

The demo program `docs/programs/sprite_multiplex.bas` shows a complete working example with multiple multiplexed sprites.

Low-level copper control via POKE

You can also control the copper by writing directly to the VGC command registers (A010–A016) and then writing the command byte to $A010.

Code	Command
$1B	**Copper Add** – add an event to the target list. P0/P1 = X (16-bit), P2 = Y, P3/P4 = register (0–15 or A000–A00F), P5 = value.
$1C	**Copper Clear** – remove all events from the target list.
$1D	**Copper Enable** – start executing the active copper list each frame.
$1E	**Copper Disable** – stop executing the copper program.
$20	**Copper List** – set target list. P0 = list index (0–127).
$21	**Copper Use** – set pending active list. P0 = list index (0–127). Takes effect at next vblank.
$22	**Copper List End** – reset target list back to the active list.

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 COPPER CLEAR
20 FOR I = 0 TO 12
30   COPPER ADD 0, I * 15, BGCOL, I + 2
40 NEXT I
50 COPPER ON
```

The loop on lines 20–40 adds 13 events, each changing the background color at Y positions 0, 15, 30, ..., 180 to successive color indices. Line 50 enables the copper. Each frame the background will display as a gradient of bands.

Tip

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.

Note

Each copper list holds up to 256 events. Events are sorted by position automatically. Multiple events at the same position are applied in register-index order. The VGC stores 128 lists; all data is host-side and consumes no 6502 address space.

Sprite Shape Slots

The VGC provides 256 shape slots (indices 0–255), each storing a 16x16 pixel sprite image (128 bytes). By default, sprite *n* uses shape slot *n*. The `SPRITESHAPE` command lets you reassign which slot a sprite renders:

```
SPRITESHAPE n, shape
```

- `n` is the sprite index (0–15).
- `shape` is the shape slot (0–255) to assign to that sprite.

Multiple sprites can share the same shape slot, so you can display several identical objects without duplicating pixel data. Shape slots beyond 15 must be populated via `DMACOPY` into sprite shape memory (space 4) or register-level commands.

`SPRITESHAPE` writes to the memory-mapped sprite register at `$A044 + n*8` (the shape field in the sprite register block).

Sprite Registers

Each of the 16 sprites has an 8-byte register block starting at $A040. The register layout per sprite is:

Offset	Field	Description
+0	XLo	X position, low byte
+1	XHi	X position, high byte
+2	YLo	Y position, low byte
+3	YHi	Y position, high byte
+4	Shape	Shape slot index (0–255)
+5	Flags	Bit 0: horizontal flip, Bit 1: vertical flip, Bit 2: enable
+6	Priority	0 = behind all, 1 = between layers, 2 = in front
+7	TransColor	Per-sprite transparent color index (0–15). Default 0 (black is transparent). Values >15 disable transparency (all 16 colors opaque).

Sprite 0 occupies A040–A047, sprite 1 occupies A048–A04F, and so on up to sprite 15 at A0B8–A0BF. These registers can be written directly with `POKE` or targeted by the copper for per-scanline updates.

SPRITESET – Per-Register Sprite Control

`SPRITESET` provides direct write access to any field in a sprite's 8-byte register block. The write is synchronised to the vertical blank interval, so no mid-frame visual glitching occurs.

```
SPRITESET sprite, field, value
```

- `sprite` is the sprite index (0–15).
- `field` is the register offset within the sprite block (0–7).
- `value` is the byte to write (0–255).

The eight field offsets map directly to the sprite register layout described in Section sprite-registers:

Field	Register
0	X position, low byte
1	X position, high byte
2	Y position, low byte
3	Y position, high byte
4	Shape slot index (0–255)
5	Flags (bit 0: H-flip, bit 1: V-flip, bit 2: enable)
6	Priority (0=behind all, 1=between layers, 2=in front)
7	Transparent color index (0–15; values >15 disable transparency)

For example, to set sprite 3's transparent color to color index 5:

```
10 SPRITESET 3, 7, 5
```

> **Note**
>
> `SPRITESET` waits for the next vblank before writing to the register. This ensures the change takes effect at a frame boundary and avoids partial-frame artifacts when changing position or shape fields.

Reading Sprite Position

`SPRITEX(n)` and `SPRITEY(n)` return the current screen position of sprite `n` by reading directly from the sprite register block.

Function	Returns
SPRITEX(n)	Current X position of sprite `n` as a signed 16-bit value.
SPRITEY(n)	Current Y position of sprite `n` as a signed 16-bit value.

Both functions combine the low and high byte registers to produce a signed 16-bit result, which correctly represents positions set by `SPRITE n,x,y`, `SPRITESET`, or the copper.

```
10 SPRITE 0, 160, 80
20 PRINT SPRITEX(0), SPRITEY(0)
```

This prints `160` and `80`, confirming the position set on line 10.

Try It Now

Try It

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.

NovaBASIC supports up to 16 hardware sprites, each 16x16 pixels with 4-bit color (16 colors per sprite). Sprites are rendered by the VGC independently of the text and graphics layers, making them ideal for game characters, projectiles, and other moving objects.

Enabling and Positioning

Each sprite has an index from 0 to 15. To use a sprite, enable it and set its position:

```
SPRITE 0,ON
SPRITE 0,100,80
```

The first command enables sprite 0. The second sets its position to x=100, y=80. Coordinates use the graphics resolution (0-319 for x, 0-199 for y).

Defining Shapes

Sprites use shape slots (0-255) stored in dedicated shape RAM. Define pixel data with `SPRITEDATA`:

Try It

```
10 REM SIMPLE ARROW SPRITE
20 SPRITEDATA 0,0,"0000000100000000"
30 SPRITEDATA 0,1,"0000001110000000"
40 SPRITEDATA 0,2,"0000011111000000"
50 SPRITEDATA 0,3,"0000000100000000"
60 SPRITEDATA 0,4,"0000000100000000"
70 SPRITE 0,ON
80 SPRITESHAPE 0,0
90 SPRITE 0,160,100
```

Each `SPRITEDATA` call defines one row (0-15) of a shape slot. The hex string encodes 16 pixels, each being a color index 0-F. Color 0 is transparent by default.

Priority Layers

Sprites render at one of three priority levels:

Priority	Constant	Rendering Order
0	Behind	Behind text and graphics
1	Between	Between background and text
2	Front	In front of everything

```
SPRITE 0,PRIORITY,2
```

Collision Detection

The VGC detects two types of collisions each frame:

- Sprite-sprite: Two enabled sprites overlap on non-transparent pixels
- Sprite-background: A sprite overlaps a non-zero graphics pixel

Read collisions with the `COLLISION()` and `BUMPED()` functions:

```
IF COLLISION(0) THEN PRINT "SPRITE␣0␣HIT␣ANOTHER␣SPRITE"
IF BUMPED(0) THEN PRINT "SPRITE␣0␣HIT␣BACKGROUND"
```

Note

Collision flags are cleared after reading. Read them once per frame and store the result if you need to check multiple times.

Flipping

Sprites can be flipped horizontally or vertically without modifying shape data:

```
SPRITE 0,FLIPX,ON
SPRITE 0,FLIPY,ON
```

Warning

Flipping applies to the rendered sprite only. The underlying shape data in shape RAM is unchanged.

Sound and Music

> Music is the arithmetic of sounds as optics is the geometry of light.
>
> – Claude Debussy

NovaBASIC includes two SID chip emulators – software recreations of the MOS 6581 Sound Interface Device made famous by the Commodore 64. Six independent voices with four waveforms, ADSR envelopes, and a programmable filter deliver authentic chiptune sound. On top of the SID sits a six-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.

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 six 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).

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.

Note

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 x 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.

Note

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

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
```

Note

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

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.

The MUSIC Engine

The music engine is a six-voice MML sequencer running on top of two SID chips. 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.

Loading and Playing Sequences

`MUSIC voice, "mml-string"`

- `voice` – voice number 1–6.
- `"mml-string"` – an MML sequence (see Section mml).

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[,...,v6]	Set voice-stealing priority for sound effects. Lower-numbered voices are stolen first. Default: 6, 5, 4, 3, 2, 1.

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.

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` (1–6), or 0 if that voice is silent.

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

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.

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).

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

Denominator	Ticks	Name
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 x 1.5 = 144 ticks (dotted quarter).

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

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.

Octave

Cmd	Description
O4	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.

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.

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.

Note

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.

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.

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.

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.

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 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.

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.

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.

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
```

Music Engine Architecture

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

Voice allocation

The music engine manages six music voices (voices 1–3 mapped to SID 1 voices 0–2; voices 4–6 mapped to SID 2 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 6 first, then 5, then 4, then 3, then 2, then 1).
3. Plays the SFX on the stolen voice; when done, restores the music voice.

The Second SID Chip

NovaBASIC includes two SID chips for a total of six voices. The first SID (D400) handles voices 1–3; the second SID (D420) handles voices 4–6. A legacy mirror at $D500 routes transparently to the second SID.

From BASIC, the second SID is fully transparent – simply use voice numbers 4–6 with the `MUSIC` command:

```
10 INSTRUMENT 0,64,2,8,12,6
20 MUSIC 1,"T140␣I0␣O4␣C4␣E4␣G4␣C5␣2"
30 MUSIC 4,"T140␣I0␣O3␣C2␣G2"
40 MUSIC PLAY
```

Voice 4 plays on the second SID chip. All instrument definitions, tempo, and loop settings apply equally to both chips.

Timing

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

ticks per frame = 96 x 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 1/8 of remaining distance per frame)
5. Filter sweep (+/-8 per frame, clamped 0–2047)

Composition Tips

Tip

- 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.

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.

Try It Now

Try It

Type and run the following program to hear a six-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 on SID 1. The music repeats until you type `MUSIC STOP` in direct mode. Try assigning additional voices (4–6) to use the second SID chip.
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"`.

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.

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.

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.

> **Note**
>
> 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.

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)
```

> **Tip**
>
> `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.

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."
```

> **Note**
>
> `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.

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.

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.

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.

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

Code	Meaning
5	Invalid name
6	End of directory

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.

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–A01F`	32 B	VGC Registers	R/W
`A040–A0BF`	128 B	Sprite Registers (16x8)	R/W
`A100–A13F`	64 B	Network Interface Controller (NIC)	R/W
`AA00–B1CF`	2000 B	Character RAM (80x25)	R/W
`B1D0–B99F`	2000 B	Color RAM (80x25)	R/W
`B9A0–B9EF`	80 B	File I/O Controller (FIO)	R/W
`BA00–BA3F`	64 B	Expansion Memory Controller (XMC)	R/W
`BA40–BA4F`	16 B	Timer Controller	R/W
`BA50–BA56`	7 B	Music Status (6 voices)	RO

Range	Size	Purpose	Access
BA60–BA7F	32 B	DMA Controller	R/W
BA80–BA9F	32 B	Blitter Controller	R/W
BC00–BFFF	1024 B	XRAM Windows (when mapped)	R/W
C000–FFFF	16 KB	ROM (NovaBASIC)	R only
D400–D41C	29 B	SID chip 1 (write-intercepted)	W only
D420–D43C	29 B	SID chip 2 (write-intercepted)	W 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's ROM initializes page $02 with EhBASIC internal vectors and the BASIC IRQ/NMI handler pointers. The layout is inherited from EhBASIC 2.22p5:

Address	Purpose
0200–0202	Ctrl-C flag, byte, and timeout
0203–0204	VEC_CC – Ctrl-C check vector
0205–0206	VEC_IN – input vector
0207–0208	VEC_OUT – output vector
0209–020A	VEC_LD – load vector
020B–020C	VEC_SV – save vector
020D–0216	IRQ handler code area (set by IRQ)
0217–0220	NMI handler code area (set by NMI)

Hardware controller base addresses are fixed constants; use the values from Appendix memmap directly in your assembly code.

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 `GET` 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
```

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.

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.

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
```

Note

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.

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 memmap.

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 atD407, voice 2 at $D40E). Filter and volume registers are at D415–D418. See Appendix memmap 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.

Copper Programming from Assembly

From BASIC, the `COPPER` keyword provides high-level access (see Chapter graphics). From assembly, write parameters to A011–A016 and the command code to $A010:

```
; Add copper event: at Y=50, X=0, set BgCol to color 5
LDA \#00 : STAA011
LDA \#00 : STAA012
LDA \#50 : STA $A013
LDA \#01 : STAA014
LDA \#00 : STAA015
LDA \#05 : STAA016
LDA \#1B : STAA010
; Enable copper
LDA \#1D : STAA010
```

The copper also supports sprite registers at A040–A0BF. Pass the absolute address as P3/P4 (16-bit little-endian). For example, to change sprite 0's Y position at scanline 80:

```
; Copper event: at Y=80, X=0, set sprite 0 YLo to 150
LDA \#00 : STAA011
LDA \#00 : STAA012
LDA \#80 : STA $A013
LDA \#42 : STAA014
LDA \#A0 : STAA015
LDA \#150 : STA $A016
LDA \#1B : STAA010
```

Here P3/P4 = $A042 (sprite 0's YLo register). See Chapter graphics for the COPPER keyword syntax and sprite multiplexing patterns.

Sprite Registers from Assembly

The sprite register block at A040–A0BF provides direct memory-mapped access to all 16 sprites. Each sprite occupies 8 bytes with a stride of 8:

Offset	Field	Size
+0	X position, low byte	1 byte
+1	X position, high byte	1 byte
+2	Y position, low byte	1 byte
+3	Y position, high byte	1 byte
+4	Shape slot index (0–255)	1 byte
+5	Flags (bit 0=hflip, bit 1=vflip, bit 2=enable)	1 byte
+6	Priority (0=behind, 1=between, 2=front)	1 byte
+7	TransColor (per-sprite transparent color index, 0–15; >15 = all opaque)	1 byte

Sprite n starts at `$A040 + n*8`. For example, sprite 3 starts at $A058. To position sprite 3 at X=200, Y=100 and enable it:

```
; Sprite 3 base = $A058
LDA \#200 : STA $A058
LDA \#00 : STAA059
LDA \#100 : STA $A05A
LDA \#00 : STAA05B
; Enable (bit 2 of flags)
LDA \#04 : STAA05D
```

Writing directly to sprite registers is faster than going through the VGC command interface and is the preferred approach from assembly code.

DMA Controller from Assembly

The DMA controller at $BA60 transfers data between six unified memory spaces. The pattern is: load parameters into registers, then write $01 to the command register to start.

```
10 REM -- DMA copy 2000 bytes from CPU $6000 to char RAM --
20 POKE $BA63, 0   : REM source space = CPU RAM
30 POKE $BA64, 1   : REM dest space = Char RAM
40 POKE $BA65, 0   : POKE $BA66, $60 : POKE $BA67, 0
50 REM source addr = $6000 (low, mid, high)
60 POKE $BA68, 0   : POKE $BA69, 0  : POKE $BA6A, 0
70 REM dest addr = 0 (start of char RAM)
80 POKE $BA6B, $D0 : POKE $BA6C, $07 : POKE $BA6D, 0
90 REM length = 2000 ($07D0)
100 POKE $BA6E, 0  : REM mode = copy (not fill)
110 POKE $BA60, 1  : REM start!
120 IF PEEK($BA61) = 1 THEN 120 : REM poll until not busy
130 IF PEEK($BA61) <> 2 THEN PRINT "Error:"; PEEK($BA62)
```

For fill mode, set bit 0 of `DmaMode` ($BA6E) and load the fill byte into `DmaFillValue` (BA6F).

Blitter from Assembly

The blitter at $BA80 performs 2D rectangular copies and fills with row stride. Set up source and destination addresses, width, height, and stride, then write $01 to `BltCmd`.

```
10 REM -- Scroll color RAM up by 1 row using blitter --
20 REM Source: row 1 (offset 80), Dest: row 0 (offset 0)
30 REM Width: 80, Height: 24, Stride: 80
40 POKE $BA83, 2  : POKE $BA84, 2   : REM src/dst = color RAM
50 POKE $BA85, 80 : POKE $BA86, 0   : POKE $BA87, 0
60 REM source offset = 80 (row 1)
70 POKE $BA88, 0  : POKE $BA89, 0   : POKE $BA8A, 0
80 REM dest offset = 0 (row 0)
90 POKE $BA8B, 80 : POKE $BA8C, 0   : REM width = 80
100 POKE $BA8D, 24 : POKE $BA8E, 0  : REM height = 24
110 POKE $BA8F, 80 : POKE $BA90, 0  : REM src stride = 80
120 POKE $BA91, 80 : POKE $BA92, 0  : REM dst stride = 80
130 POKE $BA93, 0  : REM mode = copy
```

```
140 POKE $BA80, 1   : REM start!
150 IF PEEK($BA81) = 1 THEN 150
```

Color-key mode: set bit 1 of `BltMode` ($BA93) and load the transparent color into `BltColorKey` (BA95). Source pixels matching the color key are skipped.

Network Controller from Assembly

The NIC at $A100 provides TCP networking. To connect as a client from register-level code:

```
10 REM -- Connect to 127.0.0.1 port 8080 --
20 POKE $A102, 0           : REM slot 0
30 REM Write hostname to name buffer ($A120+)
40 H$ = "127.0.0.1"
50 FOR I = 1 TO LEN(H$)
60   POKE $A11F + I, ASC(MID$(H$, I, 1))
70 NEXT I
80 POKE $A11F + LEN(H$) + 1, 0 : REM null terminate
90 POKE $A108, $90 : POKE $A109, $1F : REM port 8080
100 POKE $A100, 1   : REM connect command
110 REM Poll slot status for connected bit
120 IF (PEEK($A118) AND 1) = 0 THEN 120
```

To send data, copy the message into CPU RAM, then set `NicDmaAddrL/H` and `NicDmaLen`, and write $03 to `NicCmd`. To receive, write $04 and read `NicMsgLen` for the actual length.

The Timer Controller

The timer controller at $BA40 generates periodic IRQ interrupts, firing once every N video frames (at 60 Hz). It has no dedicated BASIC keyword – you configure it with `POKE` and handle its interrupts with `IRQ`.

Registers

Address	Name	Purpose
$BA40	TimerCtrl	Bit 0 = enable
$BA41	TimerStatus	Bit 0 = IRQ pending (reading clears)
$BA42	TimerDivL	Divisor low byte
$BA43	TimerDivH	Divisor high byte

The timer fires an IRQ every *divisor* video frames. A divisor of 60 fires once per second; a divisor of 1 fires every frame. The pending flag is cleared automatically when you read `TimerStatus`.

Example: one-second heartbeat

```
10 T = 0
20 IRQ 1000
30 POKE $BA42, 60 : POKE $BA43, 0
40 REM divisor = 60 frames = 1 second
50 POKE $BA40, 1 : REM enable
```

```
60 DO : LOOP UNTIL T >= 5
70 POKE $BA40, 0 : REM disable
80 PRINT "DONE"
90 END
1000 T = T + 1
1010 PRINT "TICK␣"; T
1020 RETIRQ
```

Note

Disable the timer (`POKE $BA40, 0`) before changing the divisor. The `SIDPLAY` command uses the timer internally; starting SID playback will reconfigure the timer divisor.

Timer from Assembly

The timer at $BA40 fires an IRQ every N video frames. To set up a 1-second timer (60 frames):

```
10 IRQ 1000
20 POKE $BA42, 60 : POKE $BA43, 0  : REM divisor = 60
30 POKE $BA40, 1                   : REM enable timer
40 GOTO 40
1000 PRINT "TICK␣";
1010 RETIRQ
```

Reading `TimerStatus` ($BA41) clears the pending IRQ flag. The timer must be disabled with `POKE $BA40, 0` before changing the divisor.

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.

DMA and Blitter

> Speed is not a luxury – it is the difference between a program that feels alive and one that feels like it is apologising.
>
> – e6502 Virtual Computer Design Notes

Every POKE loop that copies a screen buffer byte-by-byte is burning CPU cycles that your program could be spending on game logic, animation, or music. NovaBASIC includes two hardware accelerators that move data without occupying the CPU: the **DMA controller** for flat bulk transfers and the **blitter** for two-dimensional rectangular copies and fills. Both work across six unified memory spaces – CPU RAM, character RAM, color RAM, the graphics bitmap, sprite shape

memory, and expansion RAM – so data can be moved between any combination of those regions in a single command. This chapter covers both controllers, their BASIC keywords, and the memory space model they share.

The DMA Controller

Direct Memory Access (DMA) is a hardware technique that transfers a block of bytes between two memory addresses without executing a load/store loop on the CPU. The CPU hands a source address, destination address, and length to the controller and the transfer happens autonomously. From the perspective of a BASIC program the operation appears instantaneous.

The DMA controller understands six **unified memory spaces**. Each space is identified by a small integer that you pass as a parameter:

ID	Space	Size	Notes
0	CPU RAM	64 KB	Full 6502 address space
1	Character RAM	2,000 bytes	80x25 text cells
2	Color RAM	2,000 bytes	80x25 color attributes
3	Graphics Bitmap	64,000 bytes	320x200 pixels, 4-bit color
4	Sprite Shapes	32,768 bytes	256 shape slots x 128 bytes each
5	Expansion RAM	up to 512 KB	Uses current XBANK

For spaces 1 through 4, the address parameter is a zero-based byte offset into that hardware memory region – so address 0 in space 1 is the first character cell, address 80 is the start of the second row, and so on.

`DMACOPY srcSpace, srcAddr, dstSpace, dstAddr, length`

Copies *length* bytes from offset *srcAddr* in *srcSpace* to offset *dstAddr* in *dstSpace*.

```
10 REM Copy 2000 bytes from CPU RAM at $6000 to character RAM
20 DMACOPY 0, 24576, 1, 0, 2000
```

Here space 0 is CPU RAM, `24576` is $6000 decimal, space 1 is character RAM, and the destination offset is 0 (top-left cell). After line 20 executes, the 2000-byte buffer you prepared in CPU RAM has been written directly into the text screen.

```
10 REM Load graphics data from XRAM bank 0 into the bitmap
20 XBANK 0
30 DMACOPY 5, 0, 3, 0, 64000
```

Space 5 is expansion RAM. The current XBANK (set on line 20) determines which 512 KB bank is addressed. Line 30 copies all 64,000 bytes of the graphics bitmap from the start of XRAM bank 0.

`DMAFILL dstSpace, dstAddr, length, value`

Fills *length* bytes starting at offset *dstAddr* in *dstSpace* with the constant byte *value*.

```
10 REM Clear character screen with space characters (ASCII 32)
20 DMAFILL 1, 0, 2000, 32
```

```
10 REM Fill color RAM with color 7 (white-on-black attribute)
20 DMAFILL 2, 0, 2000, 7
```

Status functions

Function	Returns
DMASTATUS	0 = idle, 1 = busy, 2 = done, 3 = error
DMAERR	0 = no error; non-zero = hardware error code
DMACOUNT	Number of bytes transferred by the last operation

Note

DMACOPY and DMAFILL are asynchronous: they start the transfer and return immediately. Poll DMASTATUS until it changes from busy (1) to done (2) or error (3) when you need to wait for completion.

The Blitter

The DMA controller treats memory as a flat one-dimensional sequence of bytes. The **blitter** adds a second dimension: it understands that memory is arranged as rows of a fixed width and can copy or fill rectangular regions without touching the bytes between rows.

The key concept is **stride**: the number of bytes from the beginning of one row to the beginning of the next. For the 80-column text screen, stride is 80. For the 320-pixel-wide graphics bitmap, stride is 160 (320 pixels at 4 bits each, packed into 2 nibbles per byte). If source and destination have different strides – for example when copying a narrow tile into a wider canvas – you specify each independently.

BLITCOPY srcSpace, srcAddr, srcStride, dstSpace, dstAddr, dstStride, width, height

Copies a rectangle *width* bytes wide by *height* rows tall from the source region to the destination region.

```
10 REM Scroll the character screen up by one row.
20 REM Copy rows 1-24 (offset 80) to rows 0-23 (offset 0),
30 REM then blank the last row.
40 BLITCOPY 1, 80, 80, 1, 0, 80, 80, 24
50 DMAFILL 1, 1920, 80, 32
```

Line 40 copies 24 rows of 80 characters, shifting the entire screen up. Line 50 fills the vacated bottom row (offset 1920 = 24x80) with spaces.

```
10 REM Copy a 10x8 tile from CPU RAM at $3000 into color RAM
20 REM at column 5, row 3 (offset = 3*80+5 = 245).
30 BLITCOPY 0, 12288, 10, 2, 245, 80, 10, 8
```

The source stride is 10 because the tile is stored compactly (row-by-row, 10 bytes per row) in CPU RAM. The destination stride is 80 because color RAM is 80 columns wide.

`BLITFILL dstSpace, dstAddr, dstStride, width, height, value`

Fills a rectangle *width* bytes wide by *height* rows tall in *dstSpace* with the constant byte *value*, respecting row stride.

```
10 REM Draw a 20x5 colored rectangle starting at column 10, row 2.
20 REM offset = 2*80+10 = 170. Stride = 80.
30 BLITFILL 2, 170, 80, 20, 5, 9
```

Without the blitter this would require a nested FOR/NEXT loop: one outer loop over five rows and an inner loop over twenty columns, each issuing a POKE. `BLITFILL` replaces all of that with a single hardware call.

Color-key transparency

At the register level, bit 1 of the blitter mode register (`$BA93`) enables **color-key** mode: source bytes that equal the color key value (stored in `$BA95`) are skipped, leaving the destination byte unchanged. This is sprite-style transparency for arbitrary memory regions. The `BLITCOPY` keyword does not expose this flag directly; use `POKE $BA93, 2` before issuing a raw blitter start (`POKE $BA80, 1`) to enable it.

Status functions

Function	Returns
`BLITSTATUS`	0 = idle, 1 = busy, 2 = done, 3 = error
`BLITERR`	0 = no error; non-zero = hardware error code
`BLITCOUNT`	Number of bytes written by the last operation

Tip

Use `DMACOPY` and `DMAFILL` for flat, one-dimensional transfers – copying a saved screen buffer, filling a region of expansion RAM, or moving a block of arbitrary bytes. Use `BLITCOPY` and `BLITFILL` when your data is inherently two-dimensional and you need to respect row boundaries, such as scrolling text, drawing rectangles, or working with tile graphics. Choosing the right tool avoids writing stride arithmetic in BASIC loops.

Note

`BLITCOPY` and `BLITFILL` are also asynchronous. Use `BLITSTATUS`/`BLITERR`/`BLITCOUNT` to monitor progress and final result.

Memory Spaces Reference

The table below summarises all six space IDs accepted by both the DMA controller and the blitter. Address offsets within each space are always zero-based.

ID	Space	Size	Notes
0	CPU RAM	64 KB	Full 6502 address space. Offsets are absolute CPU addresses. Writes to ROM (C000–FFFF) are silently ignored.
1	Character RAM	2,000 bytes	80x25 text cells, row-major. Offset 0 is top-left; offset 79 is top-right; offset 80 is the start of row 1.
2	Color RAM	2,000 bytes	One color attribute byte per text cell, same layout as character RAM. Low nibble = foreground, high nibble = background.
3	Graphics Bitmap	64,000 bytes	320x200 pixels, two 4-bit pixels packed per byte (low nibble = left pixel). Row stride is 160 bytes.
4	Sprite Shapes	32,768 bytes	256 shape slots, 128 bytes each. Each slot holds a 16x16 pixel image, two pixels per byte. Sprites reference slots via shape index registers.
5	Expansion RAM	up to 512 KB	The current XBANK setting is automatically applied as the high address byte. Set `XBANK` before any DMA or blitter operation that uses space 5.

Note

When space 5 (XRAM) is the source or destination, the DMA and blitter controllers read the current XBANK register at the moment the transfer starts. Changing XBANK mid-transfer is not safe; always set the desired bank before issuing the command.

DMA and Sprite Shape Memory

`DMACOPY` can transfer shape data directly into sprite shape memory (space 4). This is the fastest way to swap sprite animations at runtime: pre-store multiple animation frames in expansion RAM and copy them on demand.

```
10 REM Copy one 128-byte shape from XRAM offset 0 to shape slot 0
20 XBANK 0
30 DMACOPY 5, 0, 4, 0, 128
```

Shape slot addresses are computed as `slot x 128`. For example, shape slot 5 starts at byte offset 640 in sprite shape space:

```
10 REM Copy shape data from XRAM into slot 5
20 DMACOPY 5, 0, 4, 640, 128
```

Tip

DMA writes to sprite shape memory are **glitch-free** – the renderer snapshots shape data at the start of each frame, so transfers can happen at any time without waiting for vblank or disabling sprites. This makes DMA-driven sprite animation safe and straightforward.

Try It Now

Try It

The program below combines all four commands to produce a continuously scrolling rainbow of color bars using only hardware accelerators – no inner loops, no per-byte POKEs.

```
10  REM -- DMA & Blitter Demo -- 20  REM Clear screen 30  DMAFILL 1,
    0, 2000, 32 40  DMAFILL 2, 0, 2000, 0 50  REM Draw 25 horizontal
    color bars, one per row 60  FOR I = 0 TO 24 70    BLITFILL 2, I
    *80, 80, 80, 1, I MOD 16 80  NEXT I 90  REM Scroll colors up
    continuously 100 BLITCOPY 2, 80, 80, 2, 0, 80, 80, 24 110 BLITFILL
     2, 24*80, 80, 80, 1, RND(1)*16 120 VSYNC 130 GOTO 100
```

Lines 30–40 clear both character and color RAM in two hardware calls. Lines 60–80 paint 25 horizontal bars of different colors by calling `BLITFILL` once per row. Lines 100–130 form an animation loop: `BLITCOPY` shifts the entire color layer up by one row, `BLITFILL` inserts a new random-color bar at the bottom, and `VSYNC` holds the frame rate at 60 Hz.
Experiments to try: - Change the fill value on line 30 to something other than 32 (ASCII space) to see different characters fill the screen – try 42 (asterisk) or 64 (at-sign). - Replace space 2 on lines 70, 100, and 110 with space 1 to scroll character codes instead of colors. - Try `DMACOPY 1, 0, 2, 0, 2000` to mirror the character RAM layout into color RAM and see the effect on the display. - Use space 5 (XRAM) as a destination: stash a full color-screen snapshot with `DMACOPY 2, 0, 5, 0, 2000` and restore it later.

Networking

The value of a network grows with the number of machines connected to it.

– Robert Metcalfe

NovaBASIC includes a built-in Network Interface Controller (NIC) that gives your programs full TCP networking from ordinary BASIC lines. You can open outbound connections to remote servers, listen for incoming connections, send and receive text messages, and build complete client/server applications – all without touching a single hardware register.

The NIC provides four independent connection **slots** numbered 0 through 3. Each slot can operate as either a client (connecting out to a remote host) or a server (listening for incoming connections). Slots are fully independent: you can have a client on slot 0 and a server on slot 1 at the same time. Messages are automatically framed so that each `NSEND` call corresponds to exactly one `NRECV$` call on the other end.

The Network Controller

The NIC implements message-oriented TCP networking. Rather than giving you a raw byte stream, it packages each message with a one-byte length prefix on the wire. This framing is invisible to BASIC: you call `NSEND` with a string and the receiver calls `NRECV$` to get that same string back intact. You never manually prefix or parse message lengths.

Key characteristics of the NIC:

- **Four independent slots** (0–3), each with its own connection state.
- **Client or server per slot** – a slot is committed to one role per connection but can switch role after being closed.
- **Message queuing** – up to 16 incoming messages may be queued per slot before the sender is blocked at the hardware level.
- **Maximum message size** of 256 bytes per send/receive.
- **DMA-backed transfers** between NIC and 6502 RAM; BASIC hides this from you entirely.

The NIC registers occupy `A100`–`A13F`. You will not normally write to those addresses directly – the BASIC networking commands issue all necessary register writes on your behalf.

> **Note**
>
> The NIC handles all TCP details: handshake, buffering, framing, and teardown. From BASIC's point of view you simply send strings and receive strings. The length-prefix framing on the wire is automatic and transparent.

Connecting as a Client

A client connection opens a TCP socket to a remote host. All four NIC commands used for client operation – `NOPEN`, `NSEND`, `NRECV$`, and `NCLOSE` – work with the same slot number from open to close.

Opening a connection: `NOPEN slot, "hostname", port`

```
NOPEN 0, "127.0.0.1", 8080
```

Establishes a TCP connection on *slot* (0–3) to *hostname* at *port* (1–65535). The hostname must be a quoted string; dotted-decimal IPv4 addresses are the most reliable form. The command blocks briefly while the TCP handshake completes; once it returns, the slot is ready to send and receive.

Sending data: `NSEND slot, "message"`

```
NSEND 0, "Hello, server!"
```

Transmits *message* through *slot*. The NIC automatically adds the one-byte length prefix before sending; the application on the far end sees only the payload. Maximum payload length is 256 bytes.

Receiving data: `NRECV$(slot)`

```
A$ = NRECV$(0)
```

Dequeues and returns the next waiting message on *slot* as a string. If no message is queued, `NRECV$` returns an empty string immediately – it does not wait. Always poll with `NREADY` first to confirm a message is available before calling `NRECV$`.

Checking for messages: `NREADY(slot)`

`NREADY(slot)` returns `-1` (true) if at least one message is waiting in the slot's receive queue, or `0` (false) if the queue is empty. Use it in a tight polling loop:

```
40 IF NOT NREADY(0) THEN 40
```

Length of last received message: `NLEN`

After a successful `NRECV$` call, `NLEN` returns the byte length of the message that was just dequeued. This is convenient when you need to know the exact payload size without calling `LEN` on the returned string.

Closing a connection: `NCLOSE slot`

```
NCLOSE 0
```

Closes the TCP connection on *slot* and releases all associated resources. Always call `NCLOSE` when you are finished with a slot, even if the remote end has already disconnected. This is required to return the slot to a clean state so it can be reused.

Complete client example

```
10 REM -- Simple TCP Client --
20 NOPEN 0, "127.0.0.1", 8080
30 NSEND 0, "Hello, server!"
40 IF NOT NREADY(0) THEN 40
50 A$ = NRECV$(0)
60 PRINT "Reply: "; A$
70 NCLOSE 0
```

Line 20 connects to a server running on the same machine at port 8080. Line 30 sends a greeting. Line 40 polls until a reply arrives. Line 50 reads the reply into `A$`. Line 60 prints it. Line 70 closes the slot.

Tip

`NREADY` and the polling loop on line 40 are the standard pattern for waiting on a response. Because NovaBASIC is single-threaded, the loop will spin tightly until data arrives. For interactive programs you can add a `VSYNC` inside the loop to yield to the display system and keep the screen responsive while waiting.

Running a Server

A server slot listens for incoming TCP connections on a local port. The workflow is slightly different from the client case because you must first bind the port, then wait for and accept a connection, before the slot behaves like a normal send/receive channel.

Binding a port: `NLISTEN slot, port`

```
NLISTEN 0, 8080
```

Begins listening for incoming connections on *port* (bound to `127.0.0.1`) using *slot*. The command returns immediately; the NIC starts accepting the OS-level TCP listen in the background. A client connecting to this port will be queued until you call `NACCEPT`.

Accepting a connection: `NACCEPT slot`

```
NACCEPT 0
```

Promotes a pending incoming connection on *slot* from the listen queue to an active connected state. You must poll `NREADY(slot)` before calling `NACCEPT` – `NREADY` returns `-1` when a client is waiting to be accepted. After `NACCEPT` returns, the slot is fully connected and you can use `NSEND` and `NRECV$` normally.

Note

`NREADY` serves double duty on a listening slot: before `NACCEPT` it signals a pending connection, and after `NACCEPT` it signals a pending message. The meaning is determined by the slot's current state.

Server loop pattern

A typical server follows this sequence:

1. `NLISTEN` to bind the port.
2. Poll `NREADY` waiting for a client to connect.
3. `NACCEPT` to bring the connection live.
4. Inner loop: poll `NREADY`, call `NRECV$`, process, `NSEND` reply. Exit when the connection drops.
5. `NCLOSE` the slot.
6. Go back to step 2 to accept the next client.

Echo server example

```
10 REM -- Echo Server --
20 NLISTEN 0, 8080
30 PRINT "Listening on port 8080..."
40 IF NOT NREADY(0) THEN 40
50 NACCEPT 0
60 PRINT "Client connected!"
70 REM Main receive/echo loop
80 IF (NSTATUS(0) AND 1) = 0 THEN 200
90 IF NOT NREADY(0) THEN 80
100 A$ = NRECV$(0)
110 L = NLEN
120 PRINT "Received (" ; L; " bytes): "; A$
130 NSEND 0, A$
140 GOTO 80
200 PRINT "Client disconnected."
210 NCLOSE 0
220 GOTO 30
```

Line 20 binds port 8080. Lines 30–40 wait for a client. Line 50 accepts it. The inner loop at lines 80–140 checks that the connection is still alive (bit 0 of `NSTATUS`), polls for incoming data, reads each message, prints the length and content, and echoes it straight back. When the connection drops, execution falls through to line 200, the slot is closed, and the server loops back to listen for the next client.

Status and Error Handling

`NSTATUS(slot)` bitmask

`NSTATUS(slot)` returns a byte whose bits describe the current state of *slot*:

Bit	Name	Meaning
0	Connected	An active TCP connection exists on this slot.
1	DataReady	At least one message is waiting in the receive queue.
2	SendReady	The slot is ready to accept an outbound message (normally always set when Connected).
3	Error	A connection or I/O error has occurred on this slot.
4	RemoteClosed	The remote peer has closed its end of the connection.

Testing individual bits

Use the `AND` operator to isolate a specific bit:

```
REM Test bit 0 (Connected)
IF (NSTATUS(0) AND 1) = 0 THEN PRINT "Disconnected"
```

```
REM Test bit 4 (RemoteClosed)
IF (NSTATUS(0) AND 16) <> 0 THEN GOSUB 900
```

Detecting disconnection

There are two ways the remote end can disappear:

- **Clean close** – the remote side called a close equivalent. Bit 4 (*RemoteClosed*) is set and bit 0 (*Connected*) clears.
- **Abrupt drop** – the network connection is lost without a graceful close. Bit 3 (*Error*) may be set; bit 0 clears.

In either case, when bit 0 drops to zero the connection is gone. Check `(NSTATUS(slot) AND 1) = 0` as your primary disconnection test, as shown in the echo server example above.

Cleaning up after disconnection

Regardless of which party initiated the close, always call `NCLOSE slot` once you detect a disconnection. This returns the slot to the idle state and allows `NLISTEN` or `NOPEN` to be called on it again.

Warning

Always check `NSTATUS` before calling `NSEND`. If bit 0 (*Connected*) is clear, the connection has dropped and `NSEND` will fail silently – your message will be discarded without any error being reported to BASIC. Test the connection state at the top of every send/receive loop.

Try It Now

Try It

Type and run the echo server program from Section nic-server:

```
10 REM -- Echo Server --
20 NLISTEN 0, 8080
30 PRINT "Listening on port 8080..."
40 IF NOT NREADY(0) THEN 40
50 NACCEPT 0
60 PRINT "Client connected!"
70 REM Main receive/echo loop
80 IF (NSTATUS(0) AND 1) = 0 THEN 200
90 IF NOT NREADY(0) THEN 80
100 A$ = NRECV$(0)
110 L = NLEN
120 PRINT "Received (";  L;  " bytes): "; A$
130 NSEND 0, A$
140 GOTO 80
200 PRINT "Client disconnected."
210 NCLOSE 0
220 GOTO 30
```

The server will print `LISTENING ON PORT 8080...` and wait. To test it, open a second emulator instance and enter the client program below, then type `RUN`:

```
10 REM -- Test Client --
20 NOPEN 0, "127.0.0.1", 8080
30 NSEND 0, "Hello from client!"
40 IF NOT NREADY(0) THEN 40
50 A$ = NRECV$(0)
60 PRINT "Echo: "; A$
70 NCLOSE 0
```

The server window should print the message and its byte length. The client window should print `ECHO: HELLO FROM CLIENT!`. After the client closes the connection the server will print `CLIENT DISCONNECTED.` and return to listening – ready for the next client.

Experiments to try:

- Modify the echo server to prefix every reply with `"ECHO: "` before sending it back.
- Change the server to count messages received and print the count alongside each one.
- Open slot 1 in the client program with a second `NOPEN` call and send a different message on each slot simultaneously to observe independent operation.
- Extend the client into a loop that sends ten numbered messages and prints each echo, then closes the connection.

Command Reference

BITCLR

Clear bits: `mem[addr] = mem[addr] AND NOT mask`.

BITSET

Set bits: `mem[addr] = mem[addr] OR mask`.

BITTGL

Toggle bits: `mem[addr] = mem[addr] EOR mask`.

BITTST

Return -1 if *all* masked bits are set at *addr*, else 0.

BLITCOPY

Copy rectangle: *w* bytes wide, *h* rows, with source stride *sstr* and destination stride *dstr*.

See the Dma And Blitter guide for more details.

BLITFILL

Fill rectangle with byte *val*.

See the Dma And Blitter guide for more details.

CALL

Execute machine code subroutine at 6502 address; `JSR`/`RTS` pair.

See the Assembly guide for more details.

CIRCLE

Draw a circle outline centred at (*cx*,*cy*).

See the Graphics And Display guide for more details.

CLEAR

Clear all variables and arrays; program is retained.

CLS

Clear the text screen and home the cursor.

See the Language Fundamentals guide for more details.

COLOR

Set foreground and optional background color (0–15).

See the Language Fundamentals guide for more details.

CONT

Continue execution after `STOP` or Ctrl-C.

COPPER_ADD

Add copper event: set *reg* to *val* at position (*x*,*y*). Registers: BGCOL, MODE, SCROLLX, SCROLLY, SPRX(*n*), SPRXH(*n*), SPRY(*n*), SPRYH(*n*), SPRSHAPE(*n*), SPRFLAGS(*n*), SPRPRI(*n*).

See the Graphics And Display guide for more details.

COPPER_CLEAR

Remove all events from the current target list.

See the Graphics And Display guide for more details.

COPPER_LIST

Set target list to *n* (0–127). Subsequent ADD/CLEAR edit this list. Reset target list back to the active list.

See the Graphics And Display guide for more details.

COPPER_OFF

Disable copper execution.

See the Graphics And Display guide for more details.

COPPER_ON

Enable copper execution each frame.

See the Graphics And Display guide for more details.

COPPER_USE

Switch active copper list to *n* at next vblank (double-buffering).

See the Graphics And Display guide for more details.

DATA

Embed constant values for `READ`.

DEC

Decrement a numeric variable by 1.

DEEK

Read a 16-bit word (little-endian) from a 6502 address (function).

See the Expansion Memory guide for more details.

DEF

Define a single-line user function.

DEL

Delete a saved program from disk.

See the First Session guide for more details.

DIM

Declare a single-dimension array.

DIR

List all saved `.bas` programs.

See the First Session guide for more details.

DMACOPY

Copy *len* bytes from space *ss* address *sa* to space *ds* address *da*.

See the Dma And Blitter guide for more details.

DMAFILL

Fill *len* bytes in space *ds* at address *da* with byte *val*.

See the Dma And Blitter guide for more details.

DO

Begin an indefinite loop body.

DOKE

Write a 16-bit word (little-endian) to a 6502 address.

See the Expansion Memory guide for more details.

END

Terminate program and return to direct mode.

FETCH

Copy *length* bytes from XRAM to CPU RAM (raw). Load a named XRAM block back into CPU RAM.

See the Expansion Memory guide for more details.

FILL

Draw a filled solid rectangle.

See the Graphics And Display guide for more details.

FOR

Begin a counted loop; step defaults to 1.

GCLS

Clear the graphics layer to transparent (all pixels 0).

See the Graphics And Display guide for more details.

GCOLOR

Set the graphics draw color (0–15; only the low nibble is used).

See the Graphics And Display guide for more details.

GET

Read a single keypress into variable (non-blocking).

GLOAD

Load a `.gfx` file into VGC memory. If *len* is omitted, loads the entire file.

See the First Session guide for more details.

GOSUB

Call subroutine at line; return address is stacked.

GOTO

Jump unconditionally to a line number.

GSAVE

Save VGC memory to a `.gfx` file. *space*: 0=screen, 1=color, 2=gfx bitmap, 3=sprite shapes.

See the First Session guide for more details.

IF

Conditional execution; `ELSE` is optional.

INC

Increment a numeric variable by 1.

INPUT

Display optional prompt and read from keyboard.

INSTRUMENT

Define instrument preset *id* (0–15). *wave*: 10=tri,20=saw, 40=pulse,80=noise. *a,d,s,r*: ADSR values 0–15.

See the Sound And Music guide for more details.

IRQ

Redirect the IRQ vector to a BASIC line number handler.

See the Assembly guide for more details.

LET

Assign value to variable (`LET` is optional).

LINE

Draw a straight line between two points.

See the Graphics And Display guide for more details.

LIST

Display program lines; omit range to list all.

LOAD

Load a previously saved program from disk.

See the First Session guide for more details.

LOCATE

Move the text cursor to column x (0–79), row y (0–24).

See the Language Fundamentals guide for more details.

LOOP

End loop; re-enter while *expr* is true. End loop; re-enter until *expr* becomes true.

MODE

Display mode: 0=text only, 1=graphics over text, 2=text over graphics, 3=graphics+sprites only (no text).

See the Graphics And Display guide for more details.

MUSIC

Load MML sequence into *voice* (1–6).

See the Sound And Music guide for more details.

MUSIC_LOOP

Enable looping (restart when all voices finish). Disable looping (default).

See the Sound And Music guide for more details.

MUSIC_PLAY

Start music playback.

See the Sound And Music guide for more details.

MUSIC_PRIORITY

Set voice-stealing priority for SFX.

See the Sound And Music guide for more details.

MUSIC_STOP

Stop music and silence all music voices.

See the Sound And Music guide for more details.

MUSIC_TEMPO

Set playback tempo (default 120).

See the Sound And Music guide for more details.

NACCEPT

Accept a pending incoming connection.

See the Networking guide for more details.

NCLOSE

Close a connection.

See the Networking guide for more details.

NEW

Clear program and all variables from memory.

NEXT

Advance loop variable and branch back if not done.

NLISTEN

Start listening for incoming connections.

See the Networking guide for more details.

NMI

Redirect the NMI vector to a BASIC line number handler.

See the Assembly guide for more details.

NOPEN

Connect to a remote TCP server.

See the Networking guide for more details.

NSEND

Send a message string.

See the Networking guide for more details.

NULL

Set the number of null (zero) bytes sent after each carriage return.

ON

Branch to the nth line in the list. Call the nth subroutine in the list.

PAINT

Flood-fill from seed point (x,y).

See the Graphics And Display guide for more details.

PEEK

Read an 8-bit byte from a 6502 address (function).

See the Expansion Memory guide for more details.

PLOT

Set pixel at (x,y) to the current draw color.

See the Graphics And Display guide for more details.

POKE

Write an 8-bit byte to a 6502 address.

See the Expansion Memory guide for more details.

PRINT

Output values to screen; `;` suppresses newline.

READ

Read successive values from `DATA` statements.

RECT

Draw a rectangle outline.

See the Graphics And Display guide for more details.

REM

Program comment; rest of line is ignored by interpreter.

RESET

Perform a full hardware and CPU reset.

RESTORE

Reset the `DATA` pointer, optionally to a line.

RETIRQ

Return from an IRQ handler (re-enables interrupts).

See the Assembly guide for more details.

RETNMI

Return from an NMI handler.

See the Assembly guide for more details.

RETURN

Return from the most recent `GOSUB`.

RUN

Execute program from the lowest line number.

SAVE

Save the current BASIC program to disk as `name.bas`.

See the First Session guide for more details.

SIDPLAY

Play a `.sid` file; optional *song* number (default 1).

See the Sound And Music guide for more details.

SIDSTOP

Stop SID file playback.

See the Sound And Music guide for more details.

SOUND

Play MIDI *note* (0–127) for *dur* frames (1/60 s). Optional *inst* selects instrument preset (0–15, default 0).

See the Sound And Music guide for more details.

SPRITE

Enable (show) sprite *n*. Disable (hide) sprite *n*. Set sprite *n* screen position.

See the Graphics And Display guide for more details.

SPRITEDATA

Define one row of sprite shape data (8 bytes, 16 pixels). Bytes b3–b8 are optional; omitted bytes leave those pixels unchanged.

See the Graphics And Display guide for more details.

SPRITESET

Write *value* (0–255) to register field *field* (0–7) of sprite *n* (0–15), waiting for vblank first. Fields: 0=X low, 1=X high, 2=Y low, 3=Y high, 4=shape, 5=flags, 6=priority, 7=transparent color.

See the Graphics And Display guide for more details.

SPRITESHAPE

Assign shape slot *shape* (0–255) to sprite *n* (0–15).

See the Graphics And Display guide for more details.

STASH

Copy *length* bytes from CPU RAM to XRAM (raw, no name). Store a named XRAM block from CPU RAM.

See the Expansion Memory guide for more details.

STOP

Break execution; `CONT` resumes at next statement.

SWAP

Exchange the values of two variables.

UNPLOT

Clear pixel at (x,y) to 0 (transparent).

See the Graphics And Display guide for more details.

VOLUME

Set SID master volume (0–15; low nibble only).

See the Sound And Music guide for more details.

VSYNC

Wait for the next video frame boundary (60 Hz).

See the Graphics And Display guide for more details.

WAIT

Busy-wait until `(PEEK(addr) XOR xor) AND mask` is non-zero.

WIDTH

Set the output line width in characters.

XALLOC

Allocate an unnamed XRAM block; returns offset in `XPEEK` result.

See the Expansion Memory guide for more details.

XBANK

Select the active 64 KB XRAM bank; n must be $<$ total banks.

See the Expansion Memory guide for more details.

XDEL

Delete a named XRAM block.

See the Expansion Memory guide for more details.

XDIR

List all named XRAM blocks and their sizes.

See the Expansion Memory guide for more details.

XFREE

Release a raw XRAM range from usage tracking.

See the Expansion Memory guide for more details.

XMAP

Map an XRAM page to CPU window 0–3 (addresses BC00–BFFF).

See the Expansion Memory guide for more details.

XMEM

Print XRAM bank count and page usage statistics.

See the Expansion Memory guide for more details.

XPEEK

Read one byte from XRAM at the given offset (function).

See the Expansion Memory guide for more details.

XPOKE

Write one byte to XRAM at the given offset in the active bank.

See the Expansion Memory guide for more details.

XRESET

Clear all XRAM allocation and named-block state (destructive).

See the Expansion Memory guide for more details.

XUNMAP

Unmap a CPU window (0–3).

See the Expansion Memory guide for more details.

Function Reference

ABS

Absolute value.

ASC

ASCII code of the first character.

ATN

Arctangent of n, result in radians.

BIN

Binary string representation of integer *n*.

BIN_STR

Binary string representation of integer *n*.

BLITCOUNT

Number of bytes written in last blit operation.

See the Dma And Blitter guide for more details.

BLITERR

Blitter error code from last operation.

See the Dma And Blitter guide for more details.

BLITSTATUS

Blitter status: 0=idle, 1=busy, 2=ok, 3=error.

See the Dma And Blitter guide for more details.

BUMPED

Sprite-to-background collision bitmask for sprite *n*.

See the Graphics And Display guide for more details.

CHR

Single-character string for ASCII code *n*.

CHR_STR

Single-character string for ASCII code *n*.

COLLISION

Sprite-to-sprite collision bitmask for sprite *n*.

See the Graphics And Display guide for more details.

COS

Cosine of *n* radians.

DMACOUNT

Number of bytes transferred in last DMA operation.

See the Dma And Blitter guide for more details.

DMAERR

DMA error code from last operation.

See the Dma And Blitter guide for more details.

DMASTATUS

DMA status: 0=idle, 1=busy, 2=ok, 3=error.

See the Dma And Blitter guide for more details.

EXP

e raised to the power *n*.

FRE

Free BASIC program memory in bytes (*x* is ignored).

HEX

Hexadecimal string representation of integer *n*.

HEX_STR

Hexadecimal string representation of integer *n*.

INT

Truncate toward zero to integer.

LCASE

String converted to lowercase.

LEFT

First *n* characters of string.

LEN

Length of string in characters.

LOG

Natural logarithm (ln n).

MAX

The larger of two numeric values.

MID

Substring starting at *start* (1-based), optional length.

MIN

The smaller of two numeric values.

MNOTE

Current MIDI note number on *voice* (1–6), or 0 if silent.

See the Sound And Music guide for more details.

NLEN

Length of the last received message (bytes).

See the Networking guide for more details.

NREADY

-1 if a message is ready, 0 otherwise.

See the Networking guide for more details.

NRECV

Receive next queued message as a string (max 255 chars).

See the Networking guide for more details.

NRECV_STR

Receive next queued message as a string (max 255 chars).

See the Networking guide for more details.

NSTATUS

Slot status bitmask (see Appendix memmap).

See the Networking guide for more details.

PI

Constant pi 3.14159...

PLAYING

1 if music is currently playing, 0 if stopped.

See the Sound And Music guide for more details.

POS

Current text cursor column position (x is ignored).

RIGHT

Last n characters of string.

RND

Pseudo-random number in [0, 1); n seeds or advances the sequence.

SADD

Address of the string's data in the string pool.

SGN

Sign of n: -1, 0, or 1.

SIN

Sine of n radians.

SPRITEX

Current X position of sprite n as a signed 16-bit value, read directly from the sprite registers.

See the Graphics And Display guide for more details.

SPRITEY

Current Y position of sprite n as a signed 16-bit value, read directly from the sprite registers.

See the Graphics And Display guide for more details.

SQR

Square root.

STR

Numeric value converted to a string.

STR_STR

Numeric value converted to a string.

TAN

Tangent of n radians.

TWOPI

Constant 2pi 6.28318...

UCASE

String converted to uppercase.

USR

Call user machine-code routine; pass x in FAC, return value in FAC.

VAL

Numeric value parsed from the leading digits of a string.

VARPTR

Address of a numeric or string variable in memory.

Memory Map

Address Space Overview

The e6502 virtual computer presents a flat 64 KB address space to the 6502 CPU. Coprocessor regions (VGC, NIC, FIO, XMC, DMA, Blitter) 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–A01F	32 B	Virtual Graphics Controller (VGC) registers and command interface
A040–A0BF	128 B	Sprite Registers (16 sprites x 8 bytes)
A100–A13F	64 B	Network Interface Controller (NIC) registers
AA00–B1CF	2,000 B	Character RAM (80x25 text cells)
B1D0–B99F	2,000 B	Color RAM (80x25 text cells)
B9A0–B9EF	80 B	File I/O Controller (FIO) registers
BA00–BA3F	64 B	Expansion Memory Controller (XMC) registers

Address Range	Size	Region
BA40–BA4F	16 B	Timer Controller registers
BA50–BA56	7 B	Music Status and Voice Note Readback (6 voices)
BA60–BA7F	32 B	DMA Controller registers
BA80–BA9F	32 B	Blitter Controller registers
BC00–BFFF	1,024 B	XMC Memory Windows (4 x 256-byte mapped pages)
C000–FFFF	16,384 B	ROM (NovaBASIC interpreter)
D400–D41C	29 B	SID chip registers (inside ROM range; writes intercepted)
D420–D43C	29 B	SID chip 2 registers (inside ROM range; writes intercepted)
D500–D51C	29 B	SID chip 2 mirror (transparently routes to $D420)

Note

The address range A020–A03F and A0C0–A0FF and A140–A9FF are not claimed by any coprocessor and fall through to the underlying flat RAM. The range BAA0–BBFF is similarly unallocated RAM. SID registers at D400–D43C occupy space within the ROM address range but are intercepted on write by the SID chip emulators.

VGC Register Map

The Virtual Graphics Controller occupies A000–A01F. 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. Register $A01F is the IRQ control register.

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, 3=graphics and sprites only (no text).
$A001	RegBgCol	R/W	Background color index (0–15).
$A002	RegFgCol	R/W	Default foreground color index (0–15); reset value is 1 (white).

Address	Name	Access	Description
$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	RegCursorEnable	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.

IRQ Control ($A01F)

Address	Name	Access	Description
$A01F	RegIrqCtrl	R/W	Bit 0: raster IRQ enable. When set, the VGC fires an IRQ once per video frame.

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).

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.

Code	Name	Parameters and behavior
$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.

Code	Name	Parameters and behavior
$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).

Tip

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 (32768 B; 256 slots). Auto-increment advances the address after each read or write.

Copper Commands (1B–1E, 20–22)

The copper triggers register writes at specific raster positions each frame. The VGC stores 128 independent copper lists (0–127), each holding up to 256 events. All copper state changes take effect at vblank.

Code	Name	Parameters and behavior
$1B	CmdCopperAdd	P0/P1 = X (16-bit), P2 = Y, P3/P4 = register (0–15, A000–A00F, or A040–A0BF for sprite registers), P5 = value. Adds to the target list. Replaces existing event at same position/register. Max 256 events per list.
$1C	CmdCopperClear	No parameters. Remove all events from the target list.
$1D	CmdCopperEnable	No parameters. Start executing the active copper list each frame.
$1E	CmdCopperDisable	No parameters. Stop executing copper.
$20	CmdCopperList	P0 = list index (0–127). Set target list for ADD/CLEAR.
$21	CmdCopperUse	P0 = list index (0–127). Set pending active list (swaps at next vblank).
$22	CmdCopperListEnd	No parameters. Reset target list to the currently active list.

System Reset Command ($1F)

Code	Name	Parameters and behavior
$1F	CmdSysReset	No parameters. Resets the VGC, stops both SID chips, stops the music engine, and resets the NIC. This is the command invoked by the BASIC `RESET` statement.

Copper-writable registers: RegMode (A000), RegBgCol (A001), RegScrollX (A005), RegScrollY (A006), and sprite registers A040–A0BF (all eight fields, including TransColor at offset +7).

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).

Second SID Chip (D420–D43C)

A second SID chip is mapped at D420–D43C with an identical register layout offset by 20 from the primary. A legacy mirror atD500–$D51C routes transparently to $D420.

Voice mapping: voices 1–3 use SID 1 ($D400$), $voices4 - -6useSID2$(D420). The music engine addresses all six voices via `MUSIC voice,"mml"` where *voice* is 1–6.

> **Note**
>
> The BASIC commands `INSTRUMENT`, `SOUND`, and `MUSIC` manage SID registers automatically. Direct writes to D400+ orD420+ are for advanced use only and may conflict with the music engine.

Timer Controller and Music Status

Timer Controller (BA40–BA4F)

The timer controller provides periodic interrupt generation. It is used internally by `SIDPLAY` but can also be configured from assembly for custom timing.

Address	Name	Access	Description
$BA40	TimerCtrl	R/W	Control register: bit 0 = enable.
$BA41	TimerStatus	RO	Status: bit 0 = IRQ pending (reading clears).
$BA42	TimerDivL	R/W	Divisor low byte.
$BA43	TimerDivH	R/W	Divisor high byte.

The timer fires an IRQ every *divisor* video frames (at 60 Hz). A divisor of 1 fires every frame; a divisor of 60 fires once per second. Reading `TimerStatus` clears the pending IRQ flag.

Music Status (BA50–BA56)

Address	Access	Description
$BA50	RO	Status flags: bit 0 = SFX playing, bit 1 = music playing.
$BA51	RO	Voice 1 current MIDI note (0 = silent). SID 1, voice 1.
$BA52	RO	Voice 2 current MIDI note (0 = silent). SID 1, voice 2.
$BA53	RO	Voice 3 current MIDI note (0 = silent). SID 1, voice 3.
$BA54	RO	Voice 4 current MIDI note (0 = silent). SID 2, voice 1.
$BA55	RO	Voice 5 current MIDI note (0 = silent). SID 2, voice 2.
$BA56	RO	Voice 6 current MIDI note (0 = silent). SID 2, voice 3.

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

DMA Controller Register Map

The DMA Controller occupies BA60–BA7F and provides bulk memory transfers across six unified memory spaces. Writing $01 to `DmaCmd` triggers the configured operation; poll `DmaStatus` for completion.

DMA Registers

Address	Name	Access	Description
$BA60	DmaCmd	W	Command byte; write $01 to start transfer.

Address	Name	Access	Description
$BA61	DmaStatus	RO	Status: 00=idle,01=busy, 02=ok,03=error.
$BA62	DmaErrCode	RO	Error detail code (see Appendix limits).
$BA63	DmaSrcSpace	R/W	Source memory space (0–5).
$BA64	DmaDstSpace	R/W	Destination memory space (0–5).
$BA65	DmaSrcL	R/W	Source address low byte.
$BA66	DmaSrcM	R/W	Source address mid byte.
$BA67	DmaSrcH	R/W	Source address high byte.
$BA68	DmaDstL	R/W	Destination address low byte.
$BA69	DmaDstM	R/W	Destination address mid byte.
$BA6A	DmaDstH	R/W	Destination address high byte.
$BA6B	DmaLenL	R/W	Transfer length low byte.
$BA6C	DmaLenM	R/W	Transfer length mid byte.
$BA6D	DmaLenH	R/W	Transfer length high byte.
$BA6E	DmaMode	R/W	Mode flags: bit 0 = fill mode.
$BA6F	DmaFillValue	R/W	Fill byte (used when bit 0 of DmaMode is set).
$BA70	DmaCountL	RO	Bytes transferred, low byte.
$BA71	DmaCountM	RO	Bytes transferred, mid byte.
$BA72	DmaCountH	RO	Bytes transferred, high byte.

DMA Memory Spaces

ID	Space
0	CPU RAM (64 KB)
1	VGC Character RAM (2,000 bytes)
2	VGC Color RAM (2,000 bytes)

ID	Space
3	VGC Graphics Bitmap (64,000 bytes)
4	VGC Sprite Shapes (32,768 bytes; 256 slots)
5	Expansion RAM (uses current XBANK)

Blitter Controller Register Map

The Blitter Controller occupies BA80–BA9F and provides 2D rectangular copy and fill operations with configurable row stride. It shares the same six memory spaces as the DMA controller. Writing $01 to `BltCmd` triggers the configured operation.

Blitter Registers

Address	Name	Access	Description
$BA80	BltCmd	W	Command byte; write $01 to start blit.
$BA81	BltStatus	RO	Status: 00=idle,01=busy, 02=ok,03=error.
$BA82	BltErrCode	RO	Error detail code (see Appendix limits).
$BA83	BltSrcSpace	R/W	Source memory space (0–5).
$BA84	BltDstSpace	R/W	Destination memory space (0–5).
$BA85	BltSrcL	R/W	Source base address low byte.
$BA86	BltSrcM	R/W	Source base address mid byte.
$BA87	BltSrcH	R/W	Source base address high byte.
$BA88	BltDstL	R/W	Destination base address low byte.
$BA89	BltDstM	R/W	Destination base address mid byte.
$BA8A	BltDstH	R/W	Destination base address high byte.
$BA8B	BltWidthL	R/W	Rectangle width low byte.
$BA8C	BltWidthH	R/W	Rectangle width high byte.
$BA8D	BltHeightL	R/W	Rectangle height low byte.
$BA8E	BltHeightH	R/W	Rectangle height high byte.

Address	Name	Access	Description
$BA8F	BltSrcStrideL	R/W	Source row stride low byte.
$BA90	BltSrcStrideH	R/W	Source row stride high byte.
$BA91	BltDstStrideL	R/W	Destination row stride low byte.
$BA92	BltDstStrideH	R/W	Destination row stride high byte.
$BA93	BltMode	R/W	Mode flags: bit 0 = fill, bit 1 = color-key.
$BA94	BltFillValue	R/W	Fill byte (when bit 0 of BltMode is set).
$BA95	BltColorKey	R/W	Transparent color (when bit 1 of BltMode is set).
$BA96	BltCountL	RO	Bytes written, low byte.
$BA97	BltCountM	RO	Bytes written, mid byte.
$BA98	BltCountH	RO	Bytes written, high byte.

Note

When color-key mode is active (bit 1 of `BltMode`), source pixels matching `BltColorKey` are skipped and the destination byte is left unchanged. This enables transparent sprite and tile overlays.

NIC Register Map

The Network Interface Controller occupies A100–A13F and provides message-oriented TCP networking over four connection slots. Writing a command byte to `NicCmd` triggers the operation on the slot selected by `NicSlot`.

NIC Registers

Address	Name	Access	Description
$A100	NicCmd	W	Command byte; writing triggers execution.
$A101	NicStatus	RO	Global status flags.
$A102	NicSlot	R/W	Active slot ID (0–3).
$A103	NicIrqCtrl	R/W	IRQ enable mask (one bit per slot).
$A104	NicIrqStatus	RO	IRQ pending flags (reading clears).
$A108	NicRemotePortL	R/W	Remote port low byte (for connect).
$A109	NicRemotePortH	R/W	Remote port high byte.
$A10A	NicLocalPortL	R/W	Local port low byte (for listen).

Address	Name	Access	Description
$A10B	NicLocalPortH	R/W	Local port high byte.
$A110	NicDmaAddrL	R/W	DMA RAM address low byte.
$A111	NicDmaAddrH	R/W	DMA RAM address high byte.
$A112	NicDmaLen	R/W	DMA length (1–255; $00 = 256).
$A113	NicMsgLen	RO	Last received message length.
$A118	NicSlotStatus0	RO	Slot 0 status flags.
$A119	NicSlotStatus1	RO	Slot 1 status flags.
$A11A	NicSlotStatus2	RO	Slot 2 status flags.
$A11B	NicSlotStatus3	RO	Slot 3 status flags.
A120–A13F	NicNameBuf	R/W	Hostname buffer (32 bytes ASCII).

NIC Command Codes

Code	Name	Behavior
$01	NicCmdConnect	Connect slot to remote host (hostname in NicNameBuf, port in NicRemotePortL/H).
$02	NicCmdDisconnect	Close active slot connection.
$03	NicCmdSend	Send message from CPU RAM (NicDmaAddrL/H, NicDmaLen).
$04	NicCmdRecv	Receive next queued message into CPU RAM.
$05	NicCmdListen	Start TCP server on local port (NicLocalPortL/H).
$06	NicCmdAccept	Accept pending connection on listening slot.

NIC Slot Status Bits

Bit	Name	Meaning
0	Connected	Slot has an active TCP connection.
1	DataReady	Receive queue has at least one message (or pending accept).
2	SendReady	Slot is ready to accept a send operation.
3	Error	Connection failed, I/O error, or queue overflow.
4	RemoteClosed	Remote peer closed the connection.

NIC Global Status Bits ($A101)

Bit	Name	Meaning
0	Ready	NIC is initialized and ready.
1	AnyData	At least one slot has queued data.
7	AnyError	At least one slot has an error flag set.

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.
$B9AA	FioGSpace	R/W	Graphics memory space for `GSAVE`/`GLOAD` (0=screen, 1=color, 2=gfx, 3=sprite).
$B9AB	FioGAddrL	R/W	Graphics offset, low byte.
$B9AC	FioGAddrH	R/W	Graphics offset, high byte.

Address	Name	Access	Description
$B9AD	FioGLenL	R/W	Graphics transfer length, low byte.
$B9AE	FioGLenH	R/W	Graphics transfer length, high byte.
$B9AF	FioDirType	RO	Directory entry type: 0=PRG, 1=SID.
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.
$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.

Code	Name	Behavior
$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 (1–6), 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).

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`.

Address	Name	Access	Description
$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.

Address	Name	Access	Description
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.

Code	Name	Behavior
$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.

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
0200–0202	Ctrl-C flag, byte, and timeout.
0203–0204	VEC_CC – Ctrl-C check vector.
0205–0206	VEC_IN – input device vector.
0207–0208	VEC_OUT – output device vector.
0209–020A	VEC_LD – load program vector.

Address	Purpose
020B–020C	VEC_SV – save program vector.
020D–0216	IRQ handler code area. Set by the `IRQ` statement.
0217–0220	NMI handler code area. Set by the `NMI` statement.
0221–027F	Reserved for future system use.

Note

The vector layout at 0200–020C is inherited from EhBASIC 2.22p5. Hardware controller base addresses are fixed constants; see the Address Space Overview table at the start of this appendix.

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.

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`.

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`.
`.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.

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.

Command / Feature	Limit and edge behavior
`LINE/RECT/CIRCLE/FILL` bounds	Drawing operations are clipped by the host renderer (`BlockGraphics.cs`). Portions of the shape outside the 320x200 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.
`SPRITESHAPE`	Tokenised and recognised by the ROM parser; writes the shape slot field in the sprite register block at `$A044 + n*8`.
`SPRITESET n,field,value`	Field must be 0–7 and value must be 0–255 (both byte-range). The write is deferred until the next vblank to prevent mid-frame glitches.
`SPRITEX(n)/SPRITEY(n)`	Combine the XLo/XHi (or YLo/YHi) register bytes of sprite n into a signed 16-bit result. Returns the actual current sprite position as set by `SPRITE n,x,y`, `SPRITESET`, or the copper.
`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.

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 1–6. 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`	1–6 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.
MML loops	Non-nesting. Maximum practical depth limited by string expansion.
Copper events	Maximum 256 events per list; 128 lists available. 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).

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.

Command / Feature	Limit and edge behavior
Window address space	The four CPU-visible windows occupy BC00–BFFF (4 x 256 bytes). Window 0 maps to BC00, window 1 toBD00, window 2 to BE00, window 3 toBF00.
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.
`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.

Command / Feature	Limit and edge behavior
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.

DMA and Blitter Limits

Command / Feature	Limit and edge behavior
Memory space IDs	Valid range: 0–5 (CPU RAM, Char, Color, Gfx, Sprite, XRAM). An invalid space ID sets the error code to BadSpace ($02).
24-bit addressing	All DMA and blitter addresses are 24-bit (low/mid/high byte). For XRAM (space 5), the current XBANK value is used as the high address byte automatically by the BASIC commands.
DMACOPY/DMAFILL	Length is 24-bit; zero-length transfers are rejected with BadArgs (04). Out-of-range addresses set Range (03). Writes to ROM space set WriteProt ($05).
BLITCOPY/BLITFILL	Width and height are 16-bit (1–65535 each). Zero width or height is rejected with BadArgs ($04). Stride is the byte offset between row starts; it may be larger than width.
Color-key mode (blitter)	When bit 1 of BltMode is set, source bytes matching BltColorKey are skipped. In fill mode, color-key is ignored.
Row buffer	When source and destination overlap in the same space, the blitter uses a temporary row buffer to prevent read-after-write corruption.
DMASTATUS/BLITSTATUS	Returns the raw status register value: 0=idle, 1=busy, 2=ok, 3=error. BASIC commands check status automatically and raise a function-call error if the operation fails.

Network Limits

Feature	Limit and edge behavior
Connection slots	4 slots (0–3). Each slot is independent and may be a client or server.
Message size	Maximum 256 bytes per message. Messages are length-prefixed on the wire (1 byte: 00 = 256,01–$FF = 1 -$ $-255).\|\|Receivequeue\|Upto16messagesperslot.Ifthequeueover$ $bytebuffer, null -$ $terminated).\|\|Portrange\|1 - -65535(16 -$ $bit).\|\|Connecttimeout\|10seconds.Iftheremotehostdoesnotresp$ `Returns a string of up to 255 characters. If no message is queued, returns an empty string. |` |NSTATUS(slot)| `Returns the slot status byte. Bit 4 (RemoteClosed) indicates the remote peer has disconnected; the slot should be closed with`NCLOSE. | |NREADY(slot)| `Returns -1 (true) if at least one message is in the receive queue, 0 (false) otherwise. |` |NLISTEN‘ binding

Status and Error Code Quick Reference

File I/O Controller (FIO) Codes

Value	Symbol	Meaning
$00	FioStatusIdle	No operation has been issued since last reset.
$02	FioStatusOk	Last operation succeeded.
$03	FioStatusError	Last operation failed; check error code.
$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
$00	XmcStatusIdle	No operation in progress.
$02	XmcStatusOk	Last operation succeeded.

Value	Symbol	Meaning
$03	XmcStatusError	Last operation failed; check error code.
$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.

DMA Controller Codes

Value	Symbol	Meaning
$00	DmaStatusIdle	No operation in progress.
$01	DmaStatusBusy	Transfer is executing.
$02	DmaStatusOk	Last transfer succeeded.
$03	DmaStatusError	Last transfer failed; check error code.
$00	DmaErrNone	No error.
$01	DmaErrBadCmd	Invalid command byte.
$02	DmaErrBadSpace	Invalid source or destination space ID.
$03	DmaErrRange	Address out of bounds for the given space.
$04	DmaErrBadArgs	Invalid arguments (e.g., length <= 0).
$05	DmaErrWriteProt	Destination is write-protected (ROM).

Blitter Controller Codes

Value	Symbol	Meaning
$00	BltStatusIdle	No operation in progress.
$01	BltStatusBusy	Blit is executing.
$02	BltStatusOk	Last blit succeeded.
$03	BltStatusError	Last blit failed; check error code.
$00	BltErrNone	No error.
$01	BltErrBadCmd	Invalid command byte.
$02	BltErrBadSpace	Invalid source or destination space ID.

Value	Symbol	Meaning
$03	BltErrRange	Rectangular region extends out of bounds.
$04	BltErrBadArgs	Invalid arguments (width <= 0, height <= 0).
$05	BltErrWriteProt	Destination is write-protected (ROM).

Tip

The BASIC runtime maps FIO, XMC, DMA, and Blitter 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.