P2 Debug Window Manual

See What Your Program Is Doing

June 2026

Version 1.0

Manual Organization

The nine DEBUG display windows of the Propeller 2

Iron Sheep Productions, LLC

P2 Knowledge Base Project

Contents

Copyright and License

Trademarks

Acknowledgments

Parallax Inc. and **Chip Gracey** for the Propeller 2 and its **DEBUG** display system. The reverse-engineered per-window theory-of-operations references that ground this manual were verified against the PNut v55 implementation.

Part I

Foundation

Before you open any single window, two things have to be in place: a clear picture of what a DEBUG display window *is*, and a working path from a source file to a live window on your screen. Part I covers both.

Chapter 1 lays out the shared model that every later chapter builds on — what the windows are, how a `DEBUG()` statement in your code becomes a picture on the host PC, and the vocabulary (declaration, name, feed, packet) used throughout the manual. Chapter 2 walks the loop end to end: compile with debugging enabled, run the program from the host application, and watch the window open.

Do this part once, in order, before you reach for an individual window — everything that follows assumes it.

Chapter 1: The DEBUG Display Windows

The P2 gives you nine kinds of on-screen window that draw the data your program sends. You do not wire up a display or write a rendering loop. You add a `DEBUG()` statement to your code; when the program runs, a window opens on the host PC and shows what you sent — text in a terminal, a trace on an oscilloscope, a spectrum, a piano keyboard. These are the **DEBUG display windows**, and this chapter is the shared model that every later chapter builds on.

The job of these windows is to make data you already have visible. A variable that holds a sensor reading is hard to interpret as a printed number scrolling by; fed to a SCOPE window it becomes a waveform you can read at a glance. A stream of bytes is hard to follow; fed to a LOGIC window it becomes channels you can watch. You choose the window that matches the shape of your data, send the data, and read the result.

This chapter teaches one model that applies to all nine windows: how you create a window, how you feed it, and how the feed stream is interpreted. Once you have that, each per-window chapter is just the specifics — which configuration keywords that window takes, what its data means, and which commands it adds. Read this chapter first; then go to the chapter for the window you need.

The nine windows

Every **DEBUG** display window is one of nine types. Each has a dedicated chapter that documents all of its configuration keywords, data formats, and commands. Pick the window that matches the shape of your data:

Window	What it shows	Chapter
TERM	A scrolling text terminal — status lines, variable dumps, logs	Ch 3
BITMAP	A pixel raster you draw into directly, framebuffer-style	Ch 4
PLOT	An XY plotting surface with layered drawing and sprites	Ch 5
LOGIC	A logic-analyzer trace of 1–32 digital channels	Ch 6
SCOPE	A time-domain oscilloscope of 1–8 sampled values	Ch 7
SCOPE_XY	An XY (Lissajous / phase) scope	Ch 8
FFT	A frequency-domain spectrum	Ch 9
SPECTRO	A spectrogram — frequency content over time	Ch 10
MIDI	A piano-keyboard display driven by MIDI note messages	Ch 11

These nine are the complete set the tool implements. Up to 32 display windows can run at the same time, so a single program can drive a terminal, a scope, and a plot together.

Which window for which problem

When you know the *kind* of data — or the question you are trying to answer — this maps it to the window to reach for:

You want to see...	Reach for	Chapter
Text — status lines, logs, variable dumps	TERM	3
A raw pixel image or framebuffer	BITMAP	4
A custom instrument, gauge, or drawn UI	PLOT	5
Several digital signals and their timing	LOGIC	6
A value changing over time (a waveform)	SCOPE	7
The relationship between two values (phase, Lissajous)	SCOPE_XY	8
The frequency content of a signal	FFT	9
How frequency content evolves over time	SPECTRO	10
MIDI note activity on a keyboard	MIDI	11

Create by name, feed by name

You work with a window in two steps. **First you create it; then you feed it.**

You create a window with a `DEBUG()` statement whose feed string begins with a backtick, a window type, and a name you choose:

```
1 DEBUG(`TERM Status SIZE 40 20)
```

The backtick (`` ` ``) marks the start of a display feed. `TERM` is the window type. `Status` is the name — you invent it, and it must be unique among your windows. Everything after the name configures the window; here `SIZE 40 20` makes it a 40-column by 20-row terminal. One `DEBUG()` statement like this creates one window.

From then on you address that window **by its name**, not by its type:

```
1 DEBUG(`Status "Ready.")
```

The name is the whole interface. The first statement said "make a TERM window and call it `Status`"; the second says "send this to `Status`." You can feed the same window from many places in your program, and you can feed two windows the same data by naming both. A complete minimal program:

```
CON _clkfreq = 200_000_000

PUB main()
  DEBUG(`TERM Status SIZE 40 20)   ' create a window named "Status"
  DEBUG(`Status "Ready.")          ' feed it by name
  repeat                           ' keep the program (and window) alive
```

Compile this with `pnut_ts -d` (Chapter 2 covers the setup) and a 40×20 terminal named `Status` opens and shows `Ready.`. The final `repeat` matters: when a P2 program ends, it stops sending, so keep the program running while you want to watch the window.

The feed stream is a sequence of elements

Everything after the window name — on the creation line and on every feed afterward — is a **feed stream**: a sequence of *elements* that the window reads left to right. The display parser recognizes a few element types, and the distinction between two of them is the single most important thing in this manual.

The element types are:

- **Keywords** — bare words like `SIZE`, `TITLE`, `POS`, `CLEAR`. These configure the window or issue a named command. Each window's chapter lists the keywords it understands.
- **Strings** — text in single quotes, such as `'Sawtooth'`, or in the Spin2 source written with double quotes, such as `"Ready."`. A string is shown literally.
- **Numbers** — written in decimal, hex (`$FF`), or binary (`%1010`). What a number *means* depends on context, and that is the rule you must internalize.

Values versus command codes

A number in the feed stream is interpreted one of two ways, and you control which:

- A number sent through a **formatter** is *displayed as text*. The formatters are the same `DEBUG()` output commands you use for serial output, in their value-only form: `` `udec_(x) ``, `` `uhex_(x) ``, `` `sdec_(x) ``, and so on. There are also shorthands: `` `(x) `` is short for `SDEC_` (signed decimal), `` `$(x) `` for `UHEX_` (hex), `` `%(x) `` for `UBIN_` (binary), `` `.(x) `` for `FDEC_` (floating point), and `` `#(x) `` to send the character whose code is `x`. If `x` holds 25, then `` `udec_(x) `` puts the two characters `2` and `5` into the stream.
- A **bare number** — one not wrapped in a formatter — is *raw input* the window interprets in its own way: in a TERM window it is a **command code** (cursor, color, and control); in the graphing windows it is a **data value** (a sample to plot). Either way it is not shown as the literal digits.

This is why a terminal treats a bare `13` as a newline rather than printing the digits "13": `13` is a command code. To show the number thirteen, you would send `` `udec_(13) `` or the literal string `"13"`. Carry this rule into every chapter:

> **To display a number, format it. Send it bare and the window takes it as raw input** — a command code in TERM, a plotted data value in the graphing windows. `` `udec_(temp) `` shows the value of `temp`; a bare `13` in TERM is a command code, not the text "13".

Each window assigns its own meaning to its command codes and to its raw data values — what a number does in a TERM window (cursor and color control) is not what it does in a SCOPE window (a sample value). Those meanings are documented in the per-window chapters. The element model — keywords, strings, formatted values, and bare command numbers — is the same everywhere.

Configuration versus commands

Within that element model, the keywords a window understands fall into three groups — the distinction the per-window chapters and the command reference (Appendix A) are organized around:

- **Creation-line configuration** sets the window up once, on the `DEBUG(`TYPE Name ...)` line that creates it — `SIZE`, `TITLE`, `POS`, and the rest. Most cannot be changed once the window exists.
- **Runtime commands** are sent *after* creation, in later feeds, to change the window's state or act on it: `COLOR` and `SET` on PLOT, `TRIGGER` on SCOPE, and so on. A few keywords are runtime-only and must not appear on the creation line; each window's chapter flags which.
- **Shared commands** are the handful every window understands, covered just below.

A window opens when its creation `DEBUG()` runs and stays open as long as your program keeps running. You can dismiss one explicitly with the shared `` `CLOSE `` command (below), but most programs never need to — when the program ends it stops feeding and the window stops updating, which is why the examples end in a `repeat`, to keep the program, and its windows, alive.

Commands common across windows

Most windows share a small set of named-keyword commands. You send them by name, after the window name, the same way you send data:

- `CLEAR` — clear the window's contents and reset it to wait for new data.
- `SAVE` — save the window's current image to a file on the host. Most windows accept `SAVE 'filename'`, and optionally `SAVE WINDOW 'filename'` to capture the whole window rather than just the display area. The file is a `.bmp`; the extension is appended automatically, so give the name *without* it (`'scope'`, not `'scope.bmp'`).

- `CLOSE` — close one window and free it. Most programs never need this — a window also stops when the program stops feeding it — but `` `CLOSE `` lets you dismiss a single window explicitly while the rest of the program keeps running.
- `UPDATE` — control buffered repainting. A window placed in update mode (by adding `UPDATE` to its creation line) does not redraw as data arrives; it repaints only when you feed it the `UPDATE` command. This eliminates flicker when you redraw a whole display at once. Not every window supports buffered mode — its chapter says whether it does.
- `PC_KEY` and `PC_MOUSE` — read the host keyboard and mouse back into your program, so a window can be interactive. These work across window types and share one mechanism, so they are covered together in Chapter 12.

Each window also has commands of its own — `TRIGGER` and `HOLDOFF` on SCOPE and LOGIC, the drawing commands on PLOT, and so on. Those belong to the window and are documented in its chapter.

How these differ from the single-step debugger

The P2 **DEBUG** system has two faces, and this manual covers only one of them.

These nine display windows **visualize data your program sends** with `DEBUG()`. Your program keeps running at full speed; the window draws whatever you feed it. The windows do not stop your code, do not single-step it, and do not inspect cogs or registers on their own. They show what you choose to send.

The P2 also has a **single-step debugger** — invoked by a plain `DEBUG` with no parentheses — which interrupts a cog, lets you step through instructions, and examine registers and flags. That is a different tool with a different purpose, and it is the subject of the separate *P2 Single-Step Debugger Manual*. This manual mentions it only to draw the line: when you want to *halt and step* code, you reach for the single-step debugger; when you want to *watch data flow* while code runs, you reach for these display windows.

Tooling, in one line

These windows are hosted by `pnut_term_ts`, the host application this manual uses throughout to open and draw them. The same **DEBUG** display windows are also hosted by **PNut** and by the **Spin Tools IDE** — all three environments open and draw them, so the examples work in any of them. You produce a program that drives them by compiling with `pnut_ts` using the `-d` (debug) option. Chapter 2 walks through installing and running both.

A note on high data rates

The `DEBUG()` link carries every element you send — every sample, pixel, and value — over a single serial connection, so **the link is the budget.** At the 2 Mbaud the tool uses, 8N1 framing costs

about 10 bits per byte, so the wire moves roughly **200 KB/s of raw bytes**, and after the `DEBUG ()` command and formatting overhead you can count on **~100–150 KB/s of actual payload**. Everything a window shows has to fit through that.

Most debugging fits comfortably: text, status panels, sensors read at a few Hz to a few kHz, and interactive controls all sit well under the budget. What does *not* **FIT** is a live high-bandwidth stream — full-motion video, or a full-rate ADC, RF, or audio feed. A single 320×240 color frame is already about 230 KB, more than a second's worth of link, so streaming video live is off the table.

When the data you care about is faster than the link, you do not give up — you reach for one of three strategies for living within the budget:

- **Pack** — use the streaming windows' **packed-data modes**, which unpack many samples from each long you send, moving several times more data per `DEBUG()` statement. Packing buys headroom, not an order of magnitude. Covered in Chapter 13.
- **Decimate** — send only 1-in-N samples for a live, always-updating view of a slowly-evolving signal. You trade detail for a continuous trend you can watch in real time. Detailed in Chapter 7.
- **Capture and dump** — let a tight PASM loop fill a buffer at full speed, then dump that buffer once over the slow link when a trigger fires. The capture runs at the P2's speed, not the link's; the link only carries the readout once. Detailed in Chapter 7.

You do not need any of these to get started — but knowing the budget exists, and that these three strategies live within it, is what keeps the later chapters honest about which uses a given window can really serve.

One discipline follows from the link being serial: sending output is not free. Each `DEBUG ()` **CALL** shifts its bytes out one at a time, and the cog waits while it does — even at 2 Mbaud that is thousands of clocks per message. Keep `DEBUG()` out of time-critical regions: a debug **CALL** inside a tight timing loop distorts the very timing you are trying to observe. When you must watch a fast path, capture into a buffer at full speed and dump it once afterward (the capture-and-dump strategy above) rather than printing from inside the loop.

Where to go next

Read **Chapter 2** for setup — installing the tools and getting your first window on screen. Then go to the chapter for the window you need: **TERM** (Ch 3) is the right first stop, since most debugging starts as text and the chapter exercises every part of the model you just learned. From there, choose by the shape of your data — a waveform wants **SCOPE** (Ch 7), digital channels want **LOGIC** (Ch 6), a custom instrument wants **PLOT** (Ch 5), and so on down the table above.

Chapter 2: Getting Started

This chapter takes you from a `.spin2` source file to a live **DEBUG** display window on your screen. The path is short: compile with debugging enabled, run the program from the host application, and the window opens. Everything in this manual builds on this loop, so it is worth doing once, end to end, before you move on to the individual windows.

What you need

Two things, and nothing else:

- **A P2 board** connected to your computer over USB. Any P2 board works; no shields, sensors, probes, or external wiring are required.
- **A PC running** `pnut_term_ts`, the host application that compiles your program, programs the P2, and opens the DEBUG display windows.

The compiler is `pnut_ts`, and the host application that opens the display windows is `pnut_term_ts`; this manual uses that pair throughout. The same **DEBUG** display windows are also hosted by **PNut** and by the **Spin Tools IDE**, so the examples work in those environments too — just confirm your environment runs the DEBUG link at 2 Mbaud (see Chapter 13). Every example in this manual runs on a bare board with the USB cable as its only connection.

Compiling with debugging

DEBUG output is not part of your program unless you ask for it. By default the compiler strips every `DEBUG()` statement, so a release build carries no debugging overhead. To keep the DEBUG statements — and the display windows they drive — you compile with the `-d` flag:

RUN THIS

```
pnut_ts -d myprogram.spin2
```

`-d` (equivalently `--DEBUG`) tells `pnut_ts` to compile the DEBUG statements into the binary instead of discarding them. Without it, the `DEBUG()` calls in your source produce no output and no windows open. This is the single most common reason a window fails to appear: the program was compiled without `-d`.

Running it

Run the compiled program from `pnut_term_ts`. It programs the P2 over USB and then listens for DEBUG output. When your program executes a `DEBUG()` statement that names a display window, `pnut_term_ts` opens that window and begins drawing into it. The window stays open and updates live as more data arrives.

DEBUG data travels from the P2 to the host over a serial link on the P2's pins 62 (transmit) and 63 (receive) — the standard programming pins — at 2 Mbaud. You do not configure any of this; it is the default, and `pnut_term_ts` is already listening on it. You only need to know the link exists, because its speed is the ceiling on how fast a window can update.

Your first window

Here is a complete program that opens a text window and prints a value:

```
CON _clkfreq = 200_000_000

PUB main() | reading
  DEBUG(`TERM Status SIZE 30 5)  ' create a 30x5 text window named "Status"
  reading := 42
  DEBUG(`Status "Reading: " `udec_(reading) 13)   ' feed it by name
  repeat                                          ' keep the window open
```

Two `DEBUG()` statements, two distinct jobs:

- The first **creates** a window. The first token after the backtick is the window type (`TERM`); the second is a name you choose (`Status`). `SIZE 30 5` makes it 30 columns by 5 rows.
- The second **feeds** that window, addressing it by the name you gave it. The quoted string prints as-is; `` `udec_(reading) `` prints the decimal text of `reading` — the characters `42`; the bare `13` is a newline.

This create-by-name, feed-by-name model is how every window in this manual works. You declare a window once with a type and a name, then drive it by that name for the rest of the program. The window type determines what the window draws and what commands it accepts — covered chapter by chapter — but the two-step pattern never changes.

> Display values with formatters, issue commands with bare numbers. `` `udec_(x) `` shows the digits of `x`; a bare `13` is the newline command, not the text "13". The valid output formatters are `UDEC`, `SDEC`, `UHEX`, `SHEX`, and `UBIN`, each with an optional trailing `_` that suppresses the auto label. There is no bare `DEC`, `HEX`, or `BIN`.

Compile it with `pnut_ts -d`, run it from `pnut_term_ts`, and a small text window titled `Status` opens showing `Reading: 42`.

The no-hardware philosophy

You do not need anything wired to the P2 to see any window in this manual work. That is deliberate. The P2 can generate its own data in software, and a generated signal drives a display

window exactly the way a real sensor would. Throughout this manual, examples produce their own data using:

- **counters** — a variable you increment in a loop,
- **the CORDIC solver** — `QSIN` / `QROTATE` for smooth waveforms and rotations,
- **the random-number generator** — `GETRND` (or the `?` operator) for noise,
- `GETCT` — the system counter, for timing and elapsed-time measurements.

A counter feeding a text window, a CORDIC sine wave feeding a SCOPE, RNG noise feeding a LOGIC trace — each exercises the full path from `DEBUG()` to a live window with no wiring at all. This program drives a text window from two software sources, a CORDIC sine and the RNG:

```
1  CON _clkfreq = 200_000_000
2
3  PUB main() | angle, wave, noise
4    DEBUG(`TERM Signals SIZE 40 5)
5    angle := 0
6    repeat
7      wave  := qsin(1000, angle, $1000)              ' CORDIC: a sine value
8      noise := GETRND() & $FF                        ' RNG: a noise value
9      DEBUG(`Signals 1 "wave=" `sdec_(wave) "  noise=" `udec_(noise))
10     angle += $0040
11     waitms(50)
```

The `1` after the window name homes the cursor each pass, so the line updates in place. Where a real-hardware version of an example is worth showing, this manual adds it as a short optional note *after* the software-only version — but you can work through the entire manual on a bare board.

Optional DEBUG configuration symbols

The defaults above need no setup. If you want to change them, declare any of these symbols in a `CON` block and the compiler picks them up:

Symbol	Default	What it sets
`DEBUG_PIN_TX`	`62`	The P2 pin that transmits DEBUG data. Must be 62 for display windows to open.
`DEBUG_PIN_RX`	`63`	The P2 pin that receives host input (keyboard, mouse).
`DEBUG_BAUD`	`2_000_000`	The DEBUG serial baud rate.
`DEBUG_COGS`	`%11111111`	Which cogs have debugging enabled; bits 7..0 enable cogs 7..0.
`DEBUG_DELAY`	`0`	Milliseconds to wait before the program starts transmitting DEBUG output.
`DEBUG_WINDOWS_OFF`	`0`	Set non-zero to suppress all DEBUG windows after programming.

For display windows to open at all, `DEBUG_PIN_TX` must be 62 — that is the pin `pnut_term_ts` listens on. The defaults already satisfy this; the symbols exist for the cases where you must move the link or limit which cogs participate.

```
CON _clkfreq = 200_000_000

CON
  DEBUG_PIN_TX = 62
  DEBUG_PIN_RX = 63
  DEBUG_BAUD   = 2_000_000

PUB main()
  DEBUG(`TERM Status SIZE 30 5)
  DEBUG(`Status "Ready." 13)
  repeat                          ' keep the window open
```

Where to go next

You now have the loop that every chapter relies on: compile with `-d`, run from `pnut_term_ts`, address a window by the name you gave it. From here:

- **Chapter 3 — TERM** covers the text window in full: cursor positioning, command codes, color pairs, and buffered updates. Start there if you are new to the display windows.

- The graphical windows each have their own chapter — **BITMAP** (Ch 4), **PLOT** (Ch 5), **LOGIC** (Ch 6), **SCOPE** (Ch 7), **SCOPE_XY** (Ch 8), **FFT** (Ch 9), **SPECTRO** (Ch 10), and **MIDI** (Ch 11). Each follows the same create-then-feed pattern shown here.

Part II

The Windows

This is the core of the manual: one chapter for each of the nine **DEBUG** display windows, every chapter in the same shape — what the window shows, how you declare it, the data commands that feed it, the control commands that shape it, and a complete, compilable example.

The windows run from the simplest to the most specialized. TERM (Chapter 3) is text — the window you reach for first. BITMAP and PLOT (Chapters 4 and 5) put pixels and vectors on a canvas. LOGIC, SCOPE, and SCOPE_XY (Chapters 6 through 8) visualize signals — logic traces, a waveform over time, and one value plotted against another. FFT and SPECTRO (Chapters 9 and 10) move into the frequency domain — a spectrum, and then a scrolling waterfall. MIDI (Chapter 11) lights an on-screen piano keyboard from note messages.

Each chapter stands on its own — go straight to the window you need. But the nine share one grammar, so once you have driven a single window you already have most of what it takes to drive any of the others.

Chapter 3: The TERM Window

Text Output

The TERM window is the text terminal of the P2 **DEBUG** system. You send it characters and strings; it shows them in a scrolling grid, the way a classic console does. It is the window you reach for first, because most debugging starts as text: a status line, a variable dump, a running log of what your program is doing.

You create one TERM window per `DEBUG(`TERM ...)` declaration, give it a name, and from then on you address it by that name. This chapter covers everything the window does — creating it, sending text, positioning the cursor, using color, and controlling when the display updates.

> Keyboard and mouse input (`PC_KEY`, `PC_MOUSE`) work in the TERM window too, but they share one mechanism across every window type, so they are covered together in Chapter 12. This chapter is about output.

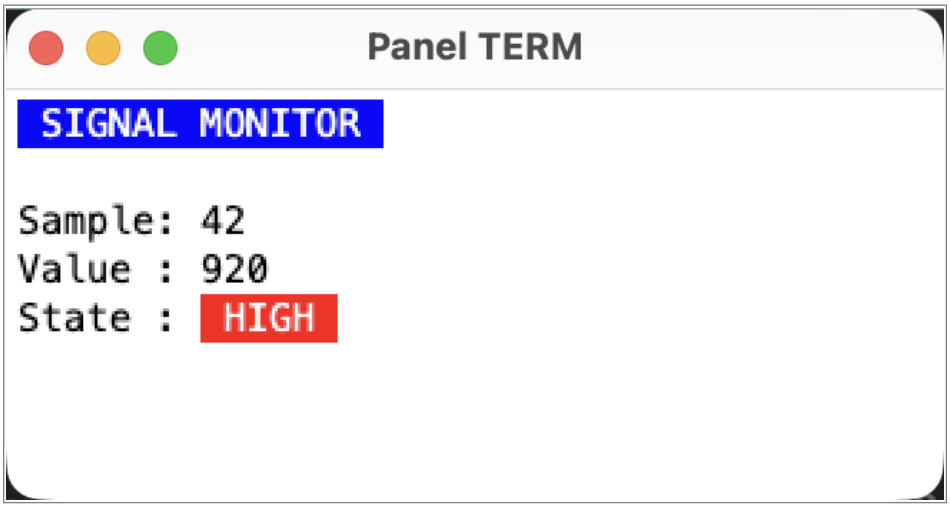

Figure 3.1. The TERM window as a positioned status dashboard.

Creating a TERM window

You create and configure a window in a single `DEBUG` statement. The first token after the backtick is the window type (`TERM`); the second is a name you choose. You feed the window afterward by that name:

```
1 PUB main()
2   DEBUG(`TERM Status SIZE 40 20)      ' create a 40x20 window named "Status"
3   DEBUG(`Status "Ready.")             ' feed it by name
```

The configuration keywords you can add to the creation line:

Keyword	Arguments	Default	What it sets
`TITLE`	`'text'`	`TERM`	The window's title-bar text
`POS`	`left top`	auto	Screen position of the window, in pixels
`SIZE`	`cols rows`	`40 20`	Grid size; each is **1–256**
`TEXTSIZE`	`points`	`10`	Font size; the window sizes itself to fit
`COLOR`	8 values	see below	Four foreground/background color pairs
`BACKCOLOR`	`rgb`	black	The canvas background — the fill used for clear and scroll (not the per-character background)
`UPDATE`	—	off	Enables buffered mode (see "Controlling updates")
`HIDEXY`	—	off	Hides the coordinate readout

`SIZE` is the one you will set most often. The grid is measured in characters, not pixels — the window computes its pixel size from the font. A `SIZE 80 25` window gives you a classic 80-column console.

Sending text

Once the window exists, everything you send by its name is rendered at the cursor, left to right, top to bottom. You can send string literals and you can send the value of a variable:

```
1 DEBUG(`Status "Temperature: " `udec_(temp) " C")
```

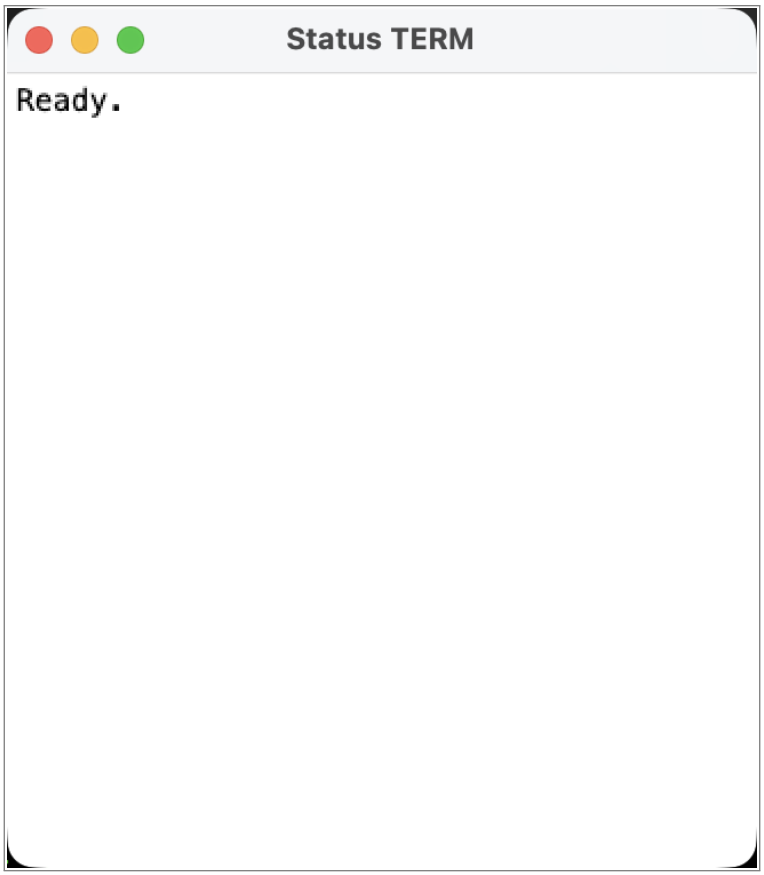

Figure 3.2. The TERM window after its first formatted value — with `temp` holding 25, the window shows `Temperature: 25 C`.

Two things are happening here, and the difference matters:

- A **quoted string** is printed as-is.
- `` `udec_(temp) `` prints the *decimal text* of `temp` — if `temp` holds 25, the window shows the characters `25`. Use the formatters (`` `udec_ ``, `` `uhex_ ``, `` `sdec_ ``, and so on) whenever you want to display a number.

There is a second, lower-level way numbers reach the window — as **command codes** — and that is the next section. The rule to carry with you: *to display a number, format it with a backtick formatter; to issue a command, send a bare number.*

Command codes

A bare number in the feed stream is not printed — it is a command. The window recognizes these codes:

Code	Command	Effect
`0`	Clear + Home	Clear the screen and move the cursor to (0, 0)
`1`	Home	Move the cursor to (0, 0) without clearing
`2`	Set column	The **next** number is the column (0-based)
`3`	Set row	The **next** number is the row (0-based)
`4`–`7`	Select color pair	Switch to color pair 0, 1, 2, or 3
`8`	Backspace	Move the cursor back one position (see caveat below)
`9`	Tab	Advance to the next 8-column tab stop
`10` / `13`	Newline	Move to the start of the next line
`32`–`255`	Character	Printable ASCII; normally you send these as strings

So to clear the screen and print a heading at row 2, column 5:

```
DEBUG(`Status 0)                  ' clear + home
DEBUG(`Status 3 2 2 5 "Heading") ' set row 2, set column 5, then print
```

Read `3 2 2 5` as two commands: `3 2` (set row to 2) and `2 5` (set column to 5). To position with a variable instead of a literal, send the value with `` `() ``:

```
DEBUG(`Status 3 `(line) 2 `(indent) "Positioned")
```

`` `(line) `` sends the *value* of `line` as the command argument — distinct from `` `udec_(line) ``, which would print it as visible digits.

Color

The TERM window holds **four color pairs**, each a foreground and a background. You select the active pair at runtime with codes `4`–`7`. The defaults are:

Pair	Code	Foreground	Background
0	`4`	Orange	Black
1	`5`	Black	Orange
2	`6`	Lime	Black
3	`7`	Black	Lime

```
DEBUG(`Status 4 "normal" 13)      ' pair 0: orange on black
DEBUG(`Status 6 "ok" 13)          ' pair 2: lime on black
```

To choose your own colors, set all eight values (four pairs, foreground then background each) on the creation line with `COLOR`. Values are `$RRGGBB`:

```
DEBUG(`TERM Log SIZE 60 20 COLOR ...
      $FF7F00 $000000 ...                 ' pair0 fg/bg
      $000000 $FF7F00 ...                 ' pair1 fg/bg
      $00FF00 $000000 ...                 ' pair2 fg/bg
      $FF0000 $000000)                    ' pair3 fg/bg
```

That gives pair 0 = orange-on-black, pair 1 = black-on-orange, pair 2 = lime-on-black, pair 3 = red-on-black — a common scheme for normal / highlighted / success / error text.

Cursor, tabs, and scrolling

- **Positioning** uses the `1` (home), `2` (set column), and `3` (set row) codes above. Columns and rows are 0-based, so the top-left cell is (0, 0).
- **Tab** (`9`) advances to the next multiple of 8 columns, printing spaces — at minimum one space, at most eight. It is how you line up columns in a table.
- **Auto-wrap**: when text reaches the right edge, the cursor moves to the start of the next line on its own.
- **Auto-scroll**: a newline on the bottom row scrolls the whole grid up one line and clears the new bottom line. A `SIZE 80 25` window used in a loop becomes a continuously scrolling log with no extra work.

```
PUB log_loop() | n
  DEBUG(`TERM Events SIZE 80 25)
  n := 0
  repeat
    DEBUG(`Events `udec_(n) ": event" 13)   ' scrolls once it fills
    n += 1
    waitms(200)
```

> **Backspace moves, it does not erase.** Code `8` steps the cursor back one cell (wrapping to the previous line at column 0) but leaves the character on screen. To replace text, reposition with `2`/`3` and overprint, or clear with `0`.

There are no ANSI escape sequences and no text attributes (bold, underline, blink). Positioning and color are done entirely with the command codes above. Line feed (`10`) and carriage return (`13`)

both mean "newline," and a CR+LF pair counts as one newline. This holds when the newline is sent as a bare command number; inside a quoted string only CR (13) breaks the line, while Chr(9) and Chr(10) render as glyphs — so text pasted into a string literal will not treat its line feeds as newlines.

Controlling updates

By default the window draws each character as it arrives — convenient for live logging. When you are redrawing a whole screen at once (a dashboard, say), that per-character drawing can flicker. Add UPDATE to the creation line to enable **buffered mode**: your output accumulates off-screen, and the window repaints only when you send the `UPDATE command.

```
PUB dashboard() | temp, press
  DEBUG(`TERM Panel SIZE 40 10 UPDATE)   ' buffered
  repeat
    temp := read_temp()
    press := read_press()
    DEBUG(`Panel 0)                       ' clear (off-screen)
    DEBUG(`Panel "Temp:  " `udec_(temp) " C" 13)
    DEBUG(`Panel "Press: " `udec_(press) " mb" 13)
    DEBUG(`Panel UPDATE)                  ' repaint once, flicker-free
    waitms(250)

PRI read_temp()  : v
  v := 25                                 ' (your sensor read here)
PRI read_press() : v
  v := 1013
```

Three more runtime keyword commands round out the set:

- `CLEAR — clears the screen and homes the cursor (identical to code 0).
- `SAVE — saves the current window image to a file on the host.
- `CLOSE — closes this window and frees its resources.

A positioned dashboard

The scrolling log and the buffered Panel both rewrite the whole display each pass. When a panel's *layout* is fixed — the labels never move, only the values change — draw the labels once, then overprint just the value fields in place with the 3 (set row) and 2 (set column) codes. Nothing scrolls, and there is no full clear.

```
CON _clkfreq = 200_000_000

PUB main() | ang, signal, count
  DEBUG(`TERM Panel SIZE 40 8)

  ' Draw the static layout once: a title and three fixed labels.
  DEBUG(`Panel 0 4 "SIGNAL MONITOR")     ' clear, pair 0, title at (0,0)
  DEBUG(`Panel 3 2 2 0 "Sample:")        ' row 2, col 0
  DEBUG(`Panel 3 3 2 0 "Value :")        ' row 3, col 0
  DEBUG(`Panel 3 4 2 0 "State :")        ' row 4, col 0

  ang := 0
  count := 0
  repeat
    signal := qsin(1000, ang, 256)       ' software-generated waveform

    ' Overprint only the value fields, each at a fixed (row, col). Trailing
    ' spaces pad to a fixed width so a shorter value erases a
    ' longer old one.
    DEBUG(`Panel 3 2 2 8 `udec_(count) "    ")
    DEBUG(`Panel 3 3 2 8 `sdec_(signal) "    ")
    if ABS signal > 800
      DEBUG(`Panel 3 4 2 8 7 "HIGH " 4)  ' pair 3 (red), then back to pair 0
    else
      ' pair 2 (lime), then back to pair 0
      DEBUG(`Panel 3 4 2 8 6 "ok   " 4)

    ang   += 4
    count += 1
    waitms(50)
```

The labels are written once; the loop touches only the three value cells, so the fields never scroll or interfere — the panel reads like a fixed instrument face.

> Pad every in-place field to a constant width (the trailing spaces above). Overprinting replaces only the characters you send, so without padding, printing `9` over `123` leaves `923`.

Where you'd use this

In computer science and computer engineering, the TERM window is the everyday tool for **systems telemetry and observability** — surfacing the internal state of a running program — and for **transaction and event logging**, a running record of what happened and when.

On an embedded project, you reach for it to show per-cog load and stack high-water marks, to inspect a peripheral's registers live, to keep running fault and event counters, or to print a state machine's current state as it advances.

Bandwidth fit: text and status panels update at human-readable rates — a few times a second is plenty — so TERM sits far inside the link budget; there is no high-rate case to temper.

Extension (real hardware): swap the synthetic `qsin` reading for a real sensor or register read, and the same panel reports live values.

Considerations

- **Pick the grid to the job.** A status panel might be `40 10`; a scrolling log wants more rows; a wide table wants more columns. Both dimensions go up to 256.
- **Buffered mode is for whole-screen redraws.** For a steadily growing log, leave it off — real-time drawing is simpler and the per-character cost is negligible.
- **Tabs are fixed at every 8 columns.** For arbitrary alignment, position explicitly with the `2` (set column) command instead.
- **Display values with formatters, issue commands with bare numbers.** This is the single most common mistake: `` `udec_(x) `` shows the number; a bare `13` is a newline, not the text "13".

Try it

Start with the dashboard example above. Then: switch it to color — print the label in pair 0 and the value in pair 2, and turn the value red (pair 3) when it crosses a threshold. You will have a live, color-coded status panel in a dozen lines, and you will have used creation config, command codes, color pairs, and buffered updates together.

Chapter 4: The BITMAP Window

Pixel Raster

The BITMAP window is the framebuffer of the P2 **DEBUG** system. You send it pixel values; it writes them straight into a bitmap canvas, one pixel per value, and shows the result. It is the window you reach for when your data *is* an image — a camera frame, a computed pattern, a sensor field, a scrolling persistence display — or whenever you want to put colored pixels on screen under direct control.

BITMAP displays pixel data and nothing else. It has no drawing primitives: there is no line, box, circle, or text command in this window. You give it numbers, and each number becomes a pixel at the current position. How those numbers are interpreted as color is set by the window's *color mode*; where each pixel lands is governed by the *trace* pattern. This chapter covers both, along with the control commands that position, scroll, clear, and save the canvas.

> Keyboard and mouse input (`PC_KEY`, `PC_MOUSE`) work in the BITMAP window, but they share one mechanism across every window type, so they are covered together in Chapter 12. This chapter is about output.

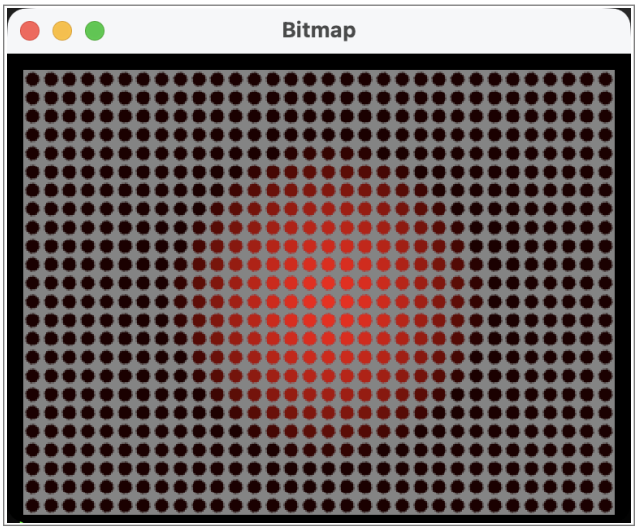

Figure 4.1. The BITMAP window showing a synthetic 32×24 thermal-array heatmap.

Creating a BITMAP window

You create and configure the window in a single `DEBUG` statement. The first token after the backtick is the window type (`BITMAP`); the second is a name you choose. You feed the window afterward by that name:

```
PUB main()
  ' create a 256x256 true-color canvas
  DEBUG(`BITMAP Img SIZE 256 256 RGB24)
  DEBUG(`Img `($FF0000))                    ' write one red pixel
```

The configuration keywords you can add to the creation line:

Keyword	Arguments	Default	What it sets
TITLE	'text'	BITMAP	The window's title-bar text
POS	left top	auto	Screen position of the window, in pixels
SIZE	width height	256 256	Canvas size in pixels; each is **1–2048**
DOTSIZE	x [y]	1 1	Pixel magnification for sparse mode; each is **1–256**
SPARSE	color	off	Enable sparse mode; sets the grid-border color
color mode	(varies)	RGB24	One of the 19 color-mode keywords (see below)
LUTCOLORS	up to 256 rgb	(none)	Define the palette for the LUT modes
TRACE	mode	0	Scan/scroll pattern, **0–15** (see "Trace patterns")
RATE	count	one scan line	Pixels written between display refreshes
LONGS_1BIT … BYTES_4BIT	—	off	Packed pixel format (see "Packed pixel data")
UPDATE	—	off	Enables manual update mode (see "Control commands")
HIDEXY	—	off	Hides the coordinate readout

SIZE sets the canvas in pixels — both dimensions run from 1 to 2048, so the canvas can be as large as 2048×2048. Memory for two full bitmaps is allocated at that size, so very large canvases are limited by host memory rather than by the window itself. Unlike SCOPE or FFT, BITMAP has no margins: the entire canvas is the drawable area, and pixel (0, 0) is the top-left corner.

Color modes

A pixel value is just a number until a color mode tells the window how many bits it carries and how to turn them into a color. You set the mode with one of 19 keywords on the creation line, and you can change it mid-stream (see "Control commands"). Every mode is converted internally to 24-bit RGB before it reaches the canvas.

The modes fall into four families:

Palette (LUT) modes — index into a 256-entry lookup table you supply with `LUTCOLORS`:

Mode	Bits/pixel	Colors	Index
`LUT1`	1	2	bit 0 → palette entry 0–1
`LUT2`	2	4	bits 0–1 → entry 0–3
`LUT4`	4	16	bits 0–3 → entry 0–15
`LUT8`	8	256	byte → entry 0–255

`RGB24` is the window's default color mode — not a LUT mode. If you select a LUT mode without defining a palette, the palette is uninitialized and LUT-mode pixels render as garbage — you must supply one with `LUTCOLORS`.

Luminance and RGB-intensity modes — 8-bit value mapped against a single tint color you pick with a color-tune keyword:

Mode	Meaning
`LUMA8` / `LUMA8W` / `LUMA8X`	Brightness in one color: black-base, white-base, expanded-range
`RGBI8` / `RGBI8W` / `RGBI8X`	Upper 3 bits select a color, lower 5 bits are intensity

For the LUMA modes you name the tint color after the keyword (for example, `LUMA8 GREEN`); the 8-bit value then runs that color from dark to bright. The `W` variants run from white toward the color; the `X` variants expand the value range.

HSV (hue/value) modes — pack a hue and a brightness into each pixel:

Mode	Bits/pixel	Layout
`HSV8` / `HSV8W` / `HSV8X`	8	4-bit hue, 4-bit value
`HSV16` / `HSV16W` / `HSV16X`	16	8-bit hue, 8-bit value

Hue maps around a color wheel (red → yellow → green → cyan → blue → magenta → red); value sets brightness. As with the luminance modes, `W` runs from white and `X` expands the range.

Direct RGB modes — the value *is* the color:

Mode	Bits/pixel	Layout
`RGB8`	8	3:3:2 (RRRGGGBB)
`RGB16`	16	5:6:5 (RRRRRGGGGGGBBBBB)
`RGB24`	24	8:8:8 (full color, `$RRGGBB`)

`RGB24` carries a full color in every pixel and needs no palette — it is the simplest mode to reason about when each pixel's color is computed independently. `RGB16` (5:6:5) and `RGB8` (3:3:2) trade color depth for bandwidth.

Only one color mode is active at a time. Pixels you sent earlier are already rendered and do not change when you switch modes; the new mode applies to every pixel that follows.

Defining a palette

For the LUT modes, LUTCOLORS loads the palette. It reads up to 256 RGB24 values and stores them as palette entries 0, 1, 2, and so on. Send the keyword and all of its colors in one DEBUG statement, because the window stops reading palette entries at the first non-numeric element:

```
PUB main() | x, y
  ' LUT4: a 16-color palette defined inline
  DEBUG(`BITMAP Tiles SIZE 16 16 LUT4 LUTCOLORS ...
         $000000 $202020 $400000 $004000 ...
         $000040 $404000 $004040 $400040 ...
         $808080 $C0C0C0 $FF0000 $00FF00 ...
         $0000FF $FFFF00 $00FFFF $FFFFFF)
  repeat y from 0 to 15
    repeat x from 0 to 15
      DEBUG(`Tiles `((x ^ y) & $0F))  ' each pixel is a 4-bit palette index
```

Because the pixels store palette *indices*, you can resend LUTCOLORS later to recolor the whole image without redrawing a single pixel — the canvas reinterprets its stored indices through the new palette.

Sending pixel data

Once the window exists, every number you send by its name becomes a pixel. You send a value with `() — the parentheses send the *value* of the expression, not its visible digits:

```
DEBUG(`Img `(color))      ' write one pixel = the value of `color`
DEBUG(`Img `($FF7F00))    ' write one orange pixel (RGB24)
```

Each pixel is plotted at the current position, and then the position advances to the next pixel according to the trace pattern. You do not address pixels individually in the common case; you stream them, and the trace pattern lays them out for you.

Trace patterns

The trace pattern decides where the first pixel goes and which way the position steps after each pixel. It is a 4-bit value set with TRACE: the low three bits choose one of eight scan patterns, and bit 3 turns on scrolling.

The eight scan patterns (bit 3 clear, values 0–7):

TRACE	Start corner	Step direction	At end of line
0	top-left	left → right	next line down, wrap
1	top-right	right → left	next line down, wrap
2	bottom-left	left → right	next line up, wrap
3	bottom-right	right → left	next line up, wrap
4	top-left	top → bottom	next column right, wrap
5	bottom-left	bottom → top	next column right, wrap
6	top-right	top → bottom	next column left, wrap
7	bottom-right	bottom → top	next column left, wrap

Patterns 0–3 scan horizontally (the X axis advances every pixel); patterns 4–7 scan vertically (the Y axis advances every pixel). Pattern 0 is the default and gives the familiar TV-raster layout: pixels fill left-to-right, top-to-bottom, and the position wraps back to the top after the last pixel.

Adding 8 to any pattern sets bit 3 and turns the end-of-line wrap into a **scroll**. Instead of wrapping to the opposite edge, the whole canvas shifts by one line and the vacated line is filled with the background color, so new data always lands at the leading edge:

TRACE	Behavior
8	top line, left → right, scroll **down**
10	bottom line, left → right, scroll **up**
12	left column, top → bottom, scroll **right**
14	right column, top → bottom, scroll **left**

(Values 9, 11, 13, 15 are the reverse-direction variants of these four.) A scrolling trace turns BITMAP into a chart recorder: each batch of pixels becomes a new line at the top while older lines march down the canvas.

Setting `TRACE` resets the position to the new pattern's start corner. The default `RATE` also follows from the pattern — horizontal patterns refresh every `width` pixels (one row), vertical patterns every `height` pixels (one column).

Packed pixel data

The DEBUG serial link is the bottleneck for any sizable image, so BITMAP can unpack several pixels from each number you send. You enable a packing format on the creation line or mid-stream; from then on, each number is split into multiple pixels before plotting.

The formats name the container size and the bits per pixel, running from `LONGS_1BIT` (32 one-bit pixels per long) through `BYTES_4BIT` (2 four-bit pixels per byte). `LONGS_1BIT` gives the largest bandwidth saving — 32× — and pairs naturally with `LUT1`:

```
PUB main() | x, y, packed, bit
  ' LONGS_1BIT: 32 one-bit pixels packed into each long
  DEBUG(`BITMAP Mono SIZE 32 32 LUT1 LONGS_1BIT)
  repeat y from 0 to 31
    packed := 0
    repeat x from 0 to 31
      bit := ((x ^ y) >> 2) & 1
      packed |= bit << x
    DEBUG(`Mono `(packed))    ' one long -> 32 pixels of one row
```

Choose a canvas width divisible by the pack count so each packed value lines up with a whole number of pixels; otherwise a value's pixels can straddle a line boundary.

Random-access writes with SET

When you do not want to stream, SET positions the pixel cursor directly. It takes an X and a Y (each must lie within the canvas bounds — an out-of-range coordinate is ignored, not clamped) and cancels any scrolling on the active pattern. The next pixel value you send lands at that position:

```
PUB main() | x, y, v
  DEBUG(`BITMAP Dots SIZE 128 128 RGB24 UPDATE)
  repeat 200
    x := GETRND() +// 128            ' GETRND gives the coordinates
    y := GETRND() +// 128
    v := GETRND() & $FFFFFF          ' and the color
    DEBUG(`Dots SET `(x, y) `(v))    ' place one pixel at (x, y)
  DEBUG(`Dots UPDATE)                ' show the result once
```

Use SET for scattered or non-sequential writes; use a trace pattern when the data arrives in scan order.

Control commands

These commands are sent at runtime, prefixed with a backtick, by the window's name. Several change configuration mid-stream; the rest manage the display.

Command	Arguments	Effect
color mode	(varies)	Switch the active color mode for following pixels
`LUTCOLORS`	up to 256 `rgb`	Reload the palette (LUT modes)
`TRACE`	`mode`	Switch scan/scroll pattern; resets position to its start corner
`RATE`	`count`	Set pixels written between display refreshes
`SET`	`x y`	Position the pixel cursor; cancels scrolling
`SCROLL`	`x y`	Shift the canvas by `x`, `y` pixels; fill the vacated strips
`CLEAR`	—	Fill the canvas with the background color; reset position
`UPDATE`	—	Refresh the display now (required in manual-update mode)
`SAVE`	—	Save the current canvas image to a file on the host
`CLOSE`	—	Close this window and free its resources

`SCROLL` shifts the whole canvas by a signed amount: positive `x` scrolls right, negative left; positive `y` scrolls down, negative up. Each argument ranges over ±canvas-dimension, and the strip exposed by the shift is filled with the background color. This is the manual counterpart to a scrolling trace — use it to pan an image or reposition a waveform.

`CLEAR` fills the canvas with the background color and returns the pixel position to the active pattern's start corner. Unless the window is in manual-update mode, it refreshes the display immediately.

`UPDATE` and manual-update mode work together. By default the canvas repaints automatically as pixels arrive (every `RATE` pixels). Add `UPDATE` to the creation line and the window stops repainting on its own; your pixels accumulate off-screen and appear only when you send the `` `UPDATE `` command. This gives you whole-frame, flicker-free updates — write an entire frame, then show it at once.

`RATE` tunes the automatic-update cadence: the display refreshes once every `count` pixels. A small count repaints often (smoother, slower); a large count repaints rarely (faster, choppier). The default is one scan line — `width` pixels for horizontal patterns, `height` for vertical; `RATE -1` selects a full canvas (`width × height`).

`SAVE` writes the current canvas to an image file on the host running `pnut_term_ts`.

A complete example

This program animates a **thermal-array heatmap**. A 32×24 grid stands in for a thermopile sensor like the MLX90640 — the kind of low-resolution infrared array you would point at a board to find a hot component, or at a doorway to count people. Each cell holds a temperature; the program renders a slowly drifting warm spot over a cool background. No hardware is involved

— the temperatures are computed in software — but the data has exactly the shape a real array would produce.

The canvas is only 32×24 logical cells, so each is magnified to a 12-pixel block with `DOTSIZE` and `SPARSE` (the magnified, low-resolution display the window is built for). `LUMA8 RED` maps each cell's 8-bit temperature from dark (cool) to bright (hot):

```
CON
  _clkfreq = 200_000_000
  COLS = 32                                  ' a 32x24 thermopile array
  ROWS = 24                                  '   (MLX90640-class)

PUB main() | x, y, ang, cx, cy

  ' 32x24 grid: each cell a 12px block with grid border; temperature -> tint
  DEBUG(`BITMAP Heat SIZE 32 24 DOTSIZE 12 SPARSE GRAY LUMA8 RED UPDATE)

  ang := 0
  repeat
    ' the warm spot drifts slowly across the array
    cx := 16 + qsin(10, ang, 256)
    cy := 12 + qsin(7, ang*2, 256)
    DEBUG(`Heat CLEAR)
    repeat y from 0 to ROWS-1
      repeat x from 0 to COLS-1
        DEBUG(`Heat `(cell(x, y, cx, cy)))
    DEBUG(`Heat UPDATE)
    ang += 3
    waitms(250)                              ' ~4 fps, like a real thermopile

PRI cell(x, y, cx, cy) : temp | dx, dy, d2
  ' synthetic temperature: a warm peak at (cx,cy) over a cool ambient
  dx := x - cx
  dy := y - cy
  d2 := dx*dx + dy*dy
  temp := (220 - d2*3) #> 0                   ' peak falls off with distance
  temp := (temp + 30) <# 255                  ' add ambient, clamp to a byte
```

Each frame: `CLEAR` resets the canvas, the nested loops stream all 768 cell values in raster order under the default trace, and `UPDATE` paints the finished frame at once. Moving the spot's center (`cx`, `cy`) each pass makes the hot region drift.

The size of this image is the lesson. A 32×24 array is 768 values; even sent as a full byte each and refreshed a few times a second, that is only a couple of kilobytes per second — comfortably inside the debug link's budget (Chapter 1). A small, slow sensor grid is exactly the kind of "image" the link carries well. A live *camera* frame is the opposite case: 320×240 in `RGB24` is about 230 KB, more than a second of link, so video would crawl in at well under one frame per second. When your image is a sensor field of tens or hundreds of cells at a few Hz, BITMAP is the right window; when it is full-motion video, it is not.

Where you'd use this

In computer science and computer engineering, the BITMAP window is for **computer vision and framebuffer visualization** — seeing the pixels a program produced or consumed — and for **2D scalar-field visualization**, where a grid of values is clearest as a colored field rather than a table of numbers.

On an embedded project, you reach for it to show a thermal-array heatmap (as here), to preview an LED-matrix framebuffer before it goes to the panel, to visualize a capacitive-touch grid, or to map signal strength or occupancy across a grid of sensors.

Bandwidth fit: small or slow grids — sensor arrays, LED matrices, touch grids, a few hundred to a few thousand cells at a few Hz — stream comfortably; live camera video does not **FIT** and is out.

Extension (real hardware): replace the synthetic `cell` temperatures with a real I²C read from an MLX90640 (or any sensor grid) into the same per-cell values, and the heatmap shows live infrared.

Considerations

- **Match the color mode to the data.** `RGB24` is simplest when each pixel's color is computed on its own. The LUT modes are smaller per pixel and let you recolor an image by reloading the palette. The HSV and LUMA modes map a single magnitude to a color, which suits sensor fields and heat maps.
- **Packing buys bandwidth.** A full `RGB24` frame is three bytes per pixel; the DEBUG link cannot stream large true-color frames quickly. For higher frame rates, drop to a LUT mode and a packed format (`LONGS_1BIT` with `LUT1` packs 32 pixels per long).
- **Pick the trace pattern to the layout.** Pattern 0 for raster images; a scrolling pattern (8, 10, 12, 14) for chart-recorder and persistence displays where new data enters at one edge.
- **Use manual update for whole frames.** When you redraw the entire canvas each pass, `UPDATE` mode prevents the partial-frame flicker you would see with automatic refresh. For a steadily growing image, automatic refresh with a suitable `RATE` is simpler.
- **Sparse mode is for magnified, low-resolution displays.** `DOTSIZE` with `SPARSE` draws each logical pixel as a `DOTSIZE`-square block with a grid border — useful for LED-matrix and pixel-art views — but it renders far more slowly than the 1:1 path, so keep the logical canvas small.

- **There are no drawing primitives here.** BITMAP plots pixels only. For lines, shapes, text, and sprites, use the PLOT window (Chapter 5), which shares BITMAP's color modes but adds a coordinate system and drawing commands.

Try it

Start with the heatmap example. First widen the temperature range or move the spot faster and watch the tint track it. Then switch the color mode: change `LUMA8 RED` to `HSV16` and feed each cell a hue computed from its temperature — cool cells blue, hot cells red — for the classic rainbow thermal palette. Next, make a different kind of display: drop `SPARSE`/`DOTSIZE`, set `SIZE 128 128` and `TRACE 8`, and send one row of values per pass to turn the window into a scrolling chart recorder. You will have exercised creation config, two color modes, sparse and full canvases, a scrolling trace, and manual updates in a single program.

Chapter 5: The PLOT Window

Vector Drawing Canvas

The PLOT window is a vector drawing canvas. You move a drawing cursor around a 2D surface and issue primitives — dots, lines, circles, ovals, rectangles, rounded rectangles, and rotated text — that the window renders with anti-aliasing. On top of the primitives it gives you a coordinate system you can re-origin and flip, a polar mode, per-element opacity, eight bitmap layers you can composite with CROP, and a 256-entry sprite system. It is the window you reach for when you need a custom instrument face, a graph, a geometric figure, or any picture that is not a text grid (Chapter 3, TERM) or a raw pixel buffer (Chapter 4, BITMAP).

You create one PLOT window per DEBUG(`PLOT ...) declaration, give it a name, and from then on address it by that name. This chapter covers everything the window does: creating it, the coordinate system, the drawing primitives, color and opacity, the layer/CROP/sprite system, and the update model that controls when your drawing becomes visible.

> Keyboard and mouse input (PC_KEY, PC_MOUSE) work in the PLOT window — the window can report the cursor position and the color under it — but that mechanism is shared across every window type, so it is covered in Chapter 12. This chapter is about drawing.

Creating a PLOT window

You create and configure a window in a single DEBUG statement. The first token after the backtick is the window type (PLOT); the second is a name you choose. You feed the window afterward by that name:

```
PUB main()
  DEBUG(`PLOT Canvas SIZE 512 512)  ' create a 512x512 canvas named "Canvas"
  DEBUG(`Canvas COLOR $00FF00 SET 256 256 DOT)   ' a green dot at the center
```

The configuration keywords you can add to the creation line:

Keyword	Arguments	Default	What it sets
TITLE	'text'	Plot	The window's title-bar text
POS	left top	auto	Screen position of the window, in pixels
SIZE	width height	256 256	Canvas size in pixels; each is **32–2048**
DOTSIZE	x {y}	1 1	Pixel magnification; each axis **1–256**
BACKCOLOR	rgb	black	Background fill color ($RRGGBB)
UPDATE	—	off	Enables buffered mode (see "The update model")
HIDEXY	—	off	Hides the mouse-coordinate readout

SIZE is measured in pixels, and both dimensions are clamped to the range **32–2048**. The default is 256 256. The canvas is drawn into an off-screen bitmap and stretched to fill the window, so resizing the window scales the picture rather than revealing more canvas.

DOTSIZE magnifies the canvas: a DOTSIZE 2 window renders each canvas pixel as a 2×2 block, which is useful when you want a small drawing shown large. The two axes can differ.

CARTESIAN, POLAR, and TEXTSIZE are **runtime commands, not creation-line keywords** — issue them after the window exists, not in the DEBUG(\PLOT …)creation CALL.CARTESIANANDPOLARselect the coordinate system;TEXTSIZESETS the default font size forTEXT(default10', range 6–200). They are described in the sections that follow.

The coordinate system

Every drawing primitive is placed relative to a **drawing cursor** — a current (x, y) position you move with SET, and that some primitives advance on their own. The cursor's coordinates are interpreted through the active coordinate system, which you control with ORIGIN, CARTESIAN, and POLAR.

Cartesian mode

Cartesian is the default. Out of the box the origin is the bottom-left corner, **x increases rightward, and y increases upward** — the mathematical convention. You change that with CARTESIAN:

Update Directive

```
CARTESIAN {flipy {flipx}}
```

- flipy — 0 leaves y increasing upward (default, the mathematical convention); 1 flips y to increase **downward**, the screen convention.
- flipx — 0 leaves x increasing rightward (default); 1 makes x increase **leftward**.

Both arguments are optional; sending `CARTESIAN` with no arguments returns to Cartesian mode (from polar) without changing the flips.

ORIGIN — moving the reference point

`ORIGIN` sets the point that coordinate (0, 0) maps to:

> **Update Directive**
>
> `ORIGIN {x y}`

- `ORIGIN x y` — place the origin at the pixel (x, y). `ORIGIN 256 256` centers the origin of a 512×512 canvas.
- `ORIGIN` with no arguments — place the origin at the *current cursor position*. Move the cursor with `SET`, then `ORIGIN`, and the origin is wherever the cursor was.

All subsequent coordinates are measured from the origin, so re-origining lets you draw a figure in its own local coordinates and place it anywhere.

```
1 PUB main()
2   DEBUG(`PLOT Centered SIZE 512 512)
3   DEBUG(`Centered ORIGIN 256 256)        ' (0,0) is now the canvas center
4   DEBUG(`Centered SET 100 100 DOT 6 255) ' a dot up-and-right of center
```

SET — moving the cursor

`SET` places the drawing cursor. Its two arguments are required, but they take different names depending on the active coordinate system:

> **Update Directive**
>
> ```
> SET x y          ' Cartesian: x, y
> SET rho theta    ' polar: radius, angle
> ```

`SET` does not draw anything; it positions the cursor for the next primitive. (Polar mode is covered below.)

Polar mode

POLAR switches the window so that the cursor's two coordinates are interpreted as **(rho, theta)** — radius and angle — and converted to Cartesian internally:

Update Directive

```
POLAR {twopi {theta}}
```

- twopi — the numeric value that represents one full turn (360°). The default is $1_0000_0000 ($2^{32}$). Choosing a power of two here makes the angle math convenient: with POLAR $1_0000, a full circle is 65 536, a quarter turn is $4000, and so on. A negative twopi reverses the rotation direction.
- theta — an angular offset added to every angle, which rotates the entire coordinate system.

With the origin at the canvas center, polar mode draws radial figures directly:

```
PUB main() | theta
  DEBUG(`PLOT Rose SIZE 512 512 POLAR $1_0000 BACKCOLOR $000000)
  DEBUG(`Rose ORIGIN 256 256)
  DEBUG(`Rose COLOR $00FFFF)
  DEBUG(`Rose SET 0 0)
  repeat theta from 0 to $1_0000
    DEBUG(`Rose LINE 200 `(theta) 1 255)   ' spokes of a wheel, radius 200
```

Send CARTESIAN to leave polar mode.

PRECISE — sub-pixel positioning

Coordinates are stored internally in a fixed-point format, and by default the window keeps **sub-pixel precision** (1/256 of a pixel), so anti-aliased primitives land on exact positions. PRECISE toggles this:

Update Directive

```
PRECISE
```

Each PRECISE flips between sub-pixel mode (the default) and whole-pixel mode. Whole-pixel mode aligns coordinates to integer pixels. Sub-pixel mode is the right choice for smooth curves and animation; you rarely need to change it.

> **Read coordinates through the active system.** A primitive's position is always the cursor, transformed by polar conversion (if active), then the origin offset, then the axis

flips. Set the system once with `ORIGIN` / `CARTESIAN` / `POLAR`, and every following primitive obeys it.

Drawing primitives

The window provides six geometric primitives plus text. Each is drawn at — or, for `LINE`, *to* — the current cursor, in the current color, with anti-aliasing. Where a primitive takes `linesize` and `opacity`, both are optional and default to the window's current line size and opacity.

DOT — a dot at the cursor

Update Directive

```
DOT {linesize {opacity}}
```

- `linesize` — dot diameter in pixels.
- `opacity` — alpha, `0`–`255` (`255` = opaque).

`DOT` draws at the cursor and **does not move it**.

```
' semi-transparent blue dot
DEBUG(`Canvas COLOR $0000FF SET 100 100 DOT 10 128)
```

LINE — a line to a new point

Update Directive

```
LINE x y {linesize {opacity}}
```

`LINE` draws from the current cursor to (x, y) — or (rho, theta) in polar mode — then **moves the cursor to that endpoint**. Because the cursor advances, you draw a connected path by issuing `LINE` repeatedly:

```
PUB main()
  DEBUG(`PLOT Box SIZE 512 512)
  DEBUG(`Box COLOR $FFFFFF SET 0 0)
  DEBUG(`Box LINE 100 0 LINE 100 100 LINE 0 100 LINE 0 0)  ' a square
```

- `linesize` — line thickness in pixels.
- `opacity` — alpha, `0`–`255`.

CIRCLE — a circle centered on the cursor

Update Directive

```
CIRCLE width {linesize {opacity}}
```

- `width` — diameter in pixels.
- `linesize` — outline thickness; `0` **(the default) fills the circle**, a value greater than zero draws an outline of that thickness.
- `opacity` — alpha, `0`–`255`.

```
DEBUG(`Canvas COLOR $FF0000 SET 256 256)
DEBUG(`Canvas CIRCLE 100 0 255)    ' filled red disc, diameter 100
DEBUG(`Canvas CIRCLE 120 5 255)    ' red ring, diameter 120, 5-pixel outline
```

OVAL — an ellipse centered on the cursor

Update Directive

```
OVAL width height {linesize {opacity}}
```

`width` and `height` are the horizontal and vertical diameters; `linesize` and `opacity` behave as for `CIRCLE` (`0` linesize fills).

```
' filled ellipse
DEBUG(`Canvas COLOR $00FF00 SET 256 256 OVAL 200 100 0 255)
```

BOX — a rectangle centered on the cursor

Update Directive

```
BOX width height {linesize {opacity}}
```

`width` and `height` are the rectangle dimensions; `linesize 0` fills, greater than zero outlines. The rectangle is centered on the cursor.

```
DEBUG(`Canvas COLOR $0000FF SET 100 100 BOX 80 60 0 255)  ' filled rectangle
DEBUG(`Canvas BOX 90 70 3 128)  ' outline, thickness 3
```

OBOX — a rounded rectangle centered on the cursor

Update Directive

```
OBOX width height xradius yradius {linesize {opacity}}
```

`OBOX` adds corner-radius arguments: `xradius` and `yradius` set the horizontal and vertical rounding of the corners. As with the other shapes, `linesize 0` fills and greater than zero outlines.

```
DEBUG(`Canvas COLOR $FFFF00 SET 256 256)
DEBUG(`Canvas OBOX 100 80 10 10 0 255)    ' filled, 10-pixel rounded corners
DEBUG(`Canvas OBOX 120 100 15 15 4 200)   ' outline, thickness 4
```

TEXT — a string at the cursor

Update Directive

```
TEXT {size {style {angle}}} 'string'
```

`TEXT` renders the string at the cursor, in the current text color. All three numeric arguments are optional and default to the window's current text size, style, and angle:

- `size` — font size in points.
- `style` — a style byte (below).
- `angle` — rotation in degrees, `0`–`359`. In polar mode the angle is given in `twopi` units instead.

```
PUB main()
  DEBUG(`PLOT Labels SIZE 600 400 BACKCOLOR $FFFFFF TEXTSIZE 14)
  DEBUG(`Labels COLOR $000000 SET 300 200)
  DEBUG(`Labels TEXT 'Default')           ' size 14 (the window default)
  DEBUG(`Labels TEXT 20 'Bigger')         ' size 20
  ' size 16, bold, rotated 90 degrees
  DEBUG(`Labels TEXT 16 $02 90 'Rotated')
```

The `style` byte packs weight, italic, underline, and alignment into one value:

Bits	Field	Values
0–1	Weight	0=100 (thin), 1=400 (normal), 2=700 (bold), 3=900 (heavy)
2	Italic	0=normal, 1=italic
3	Underline	0=none, 1=underline
4–5	Horizontal align	0/1=center, 2=left, 3=right
6–7	Vertical align	0/1=center, 2=top, 3=bottom

So $02 is bold, $06 is bold + italic, $0A is normal-weight + underline, and $20 left-aligns. The defaults (style $00) are thin weight, centered both ways.

You can set the text defaults independently with TEXTSIZE size, TEXTSTYLE style, and TEXTANGLE angle; a later TEXT that omits an argument uses the default you set.

Color and opacity

The PLOT window draws in 24-bit RGB. You set the active drawing color with COLOR:

Update Directive

```
COLOR rgb
```

The argument is a $RRGGBB value — for example COLOR $FF0000 is red, COLOR $00FF00 is green, COLOR $0000FF is blue. Named colors (RED, GREEN, BLUE, WHITE, BLACK, CYAN, MAGENTA, YELLOW, ORANGE, GRAY) are also accepted in the command stream. Until you set COLOR, the default draw color is cyan ($00FFFF) and the default text color is white ($FFFFFF).

BACKCOLOR rgb sets the background fill — the color CLEAR paints the canvas with. It is most often set on the creation line.

OPACITY sets the default alpha applied to primitives that do not specify their own:

Update Directive

```
OPACITY BYTE
```

The value is a byte 0–255 (values outside that range wrap to their low 8 bits, so they are not saturated), where 255 is fully opaque and lower values blend the primitive with whatever is already on the canvas. A per-primitive opacity argument (the last argument of DOT, LINE, CIRCLE, and the rest) overrides this default for that one primitive.

```
1 PUB main()
2   DEBUG(`PLOT Blend SIZE 512 512 BACKCOLOR $000000)
```

continues on next page →

↪ *continued from previous page*

```
DEBUG(`Blend COLOR $FF0000 OPACITY 128)        ' default to 50% opacity
DEBUG(`Blend SET 128 128 BOX 100 100 0 255)  ' this box overrides: opaque
DEBUG(`Blend SET 180 128 BOX 100 100)  ' this box uses the 128 default
```

`LINESIZE size` sets the default line/dot thickness used when `DOT` and `LINE` omit their `linesize` argument.

Layers, CROP, and sprites

Beyond live primitives, the PLOT window holds **eight bitmap layers** and a **256-entry sprite table**. Both let you assemble a picture from prebuilt pieces rather than redrawing primitives every frame: you composite a layer or stamp a sprite with a single command, which avoids re-issuing the geometry that built it.

> `{Spin2_v50}` **required.** `LAYER`, `CROP`, `SPRITEDEF`, and `SPRITE` are V50 additions. The source file's first line must be `{Spin2_v50}` (or later), compiled with a Spin2 v50+ `pnut_ts`; without it these commands are not recognized.

LAYER — load a bitmap into a layer

Update Directive

```
LAYER layer 'filename.bmp'
```

`LAYER` loads a Windows BMP file from the host into one of the eight layers; one BMP pixel maps to one canvas pixel with no scaling, so author the image at the exact pixel size you will display it. A 24-bit, uncompressed BMP is the safe choice.

- `layer` — the layer index, `1`–`8` (there is no layer 0).
- `filename` — a path to a file that must exist on the host and must end in `.bmp`. If the file is missing or has the wrong extension, the command is ignored.

Because the bitmap is read from the host filesystem, `LAYER` depends on a file being present on the machine running `pnut_term_ts`; it is not generated by the P2. The command form is what your P2 program sends; the artwork is supplied host-side.

CROP — composite a layer onto the canvas

`CROP` copies pixels from a loaded layer onto the main canvas. It has three forms:

Update Directive

```
CROP layer
CROP layer AUTO x y
CROP layer left top width height {x y}
```

- `CROP layer` — copy the entire layer to the canvas at (0, 0).
- `CROP layer AUTO x y` — copy the entire layer to the canvas at (x, y).
- `CROP layer left top width height {x y}` — copy a rectangular region of the layer — starting at (`left`, `top`) and `width`×`height` in size — to the canvas. The destination defaults to (`left`, `top`) and can be overridden with the optional trailing (`x`, `y`).

The copy is a pixel-for-pixel block transfer with no scaling, and in the default pixel format it is an **opaque** block copy, so each `CROP` overwrites its destination rectangle completely. That is why there is no separate "clear": you *erase by restoring* — copy clean background back over a region (the second form) or repaint the whole scene (the first form). Chapter 15 builds the sprite-sheet panel technique on these three idioms.

SPRITEDEF and SPRITE — defining and stamping sprites

A sprite is a small palette-indexed bitmap, up to 32×32 pixels, that you define once and then stamp anywhere, at any of eight orientations and any scale.

`SPRITEDEF` defines one:

Update Directive

```
SPRITEDEF id xsize ysize pixels... colors...
```

- `id` — sprite identifier, `0`–`255`.
- `xsize`, `ysize` — sprite dimensions, each `1`–`32`.
- `pixels` — `xsize x ysize` palette indices, one per pixel, in row order.
- `colors` — the palette entries the pixel bytes reference, in `$AARRGGBB` form (alpha, red, green, blue), where alpha `$00` is transparent and `$FF` is opaque. Supply **up to 256**, but only as many as your indices actually use — the parser reads color longs until the `DEBUG()` message ends, so a sprite that uses indices 0 and 1 needs just two colors.

`SPRITE` stamps a defined sprite at the cursor:

Update Directive

```
SPRITE id {orientation {scale {opacity}}}
```

- `id` — which sprite to draw, `0`–`255`.
- `orientation` — `0`–`7`, selecting one of eight flips/rotations (0 = normal, 1 = flip-X, 2 = flip-Y, 3 = 180°, 4–7 = the 90° rotations and their flips).
- `scale` — pixel magnification, `1`–`64`; each sprite pixel becomes a `scale x scale` block.
- `opacity` — an overall alpha multiplier, `0`–`255`, applied on top of each pixel's own alpha.

```
PUB main()
  DEBUG(`PLOT Scene SIZE 512 512 BACKCOLOR $000000 UPDATE)
  DEBUG(`Scene CLEAR)
  DEBUG(`Scene LAYER 1 'background.bmp')          ' host-supplied artwork
  DEBUG(`Scene CROP 1)  ' composite full background
  ' tiny 2x2 sprite
  DEBUG(`Scene SPRITEDEF 0 2 2 0 1 1 0 $00000000 $FFFFFFFF)
  DEBUG(`Scene SET 256 256 SPRITE 0 0 8 255)      ' stamp it, 8x scale
  DEBUG(`Scene UPDATE)  ' present the buffered frame
```

Layers are indexed 1–8. A common error is using layer 0; the lowest valid layer is 1, and `LAYER`/`CROP` ignore an index outside 1–8. Sprite *ids*, by contrast, start at 0 and run to 255.

Animating with a sprite

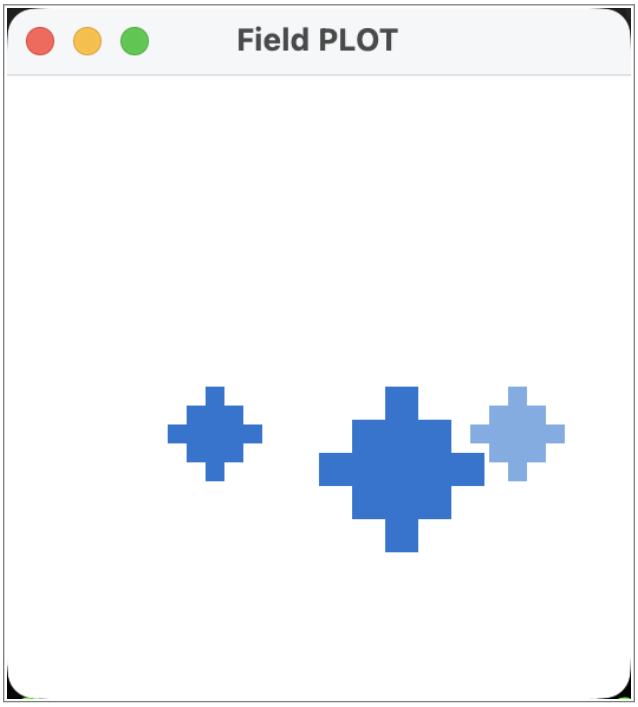

Figure 5.1. A sprite stamped at several scales in the PLOT window.

Re-stamping a sprite is cheaper than re-drawing the geometry that produced it. Define the shape once with `SPRITEDEF`, then each frame issue a single `SPRITE` command at the new position — you re-send one command, not the primitives. In buffered mode the motion is flicker-free:

```
1  CON _clkfreq = 200_000_000
2
3  PUB main() | x
4    DEBUG(`PLOT Field SIZE 256 256 BACKCOLOR $000000 UPDATE)
5    ' Define a 3x3 "blip" sprite once: index 1 = lit, index 0 = transparent.
6    ' Palette: entry 0 = transparent ($00......),
7    ' entry 1 = opaque cyan ($FF00FFFF).
8    DEBUG(`Field SPRITEDEF 0 3 3  0 1 0  1 1 1  0 1 0  $00000000 $FF00FFFF)
9
10   x := 0
11   repeat
12     DEBUG(`Field CLEAR)  ' clear the buffered canvas
13     ' stamp the sprite at (x,128), 8x
14     DEBUG(`Field SET `(x) 128 SPRITE 0 0 8 255)
15     DEBUG(`Field UPDATE)                          ' present the frame
```

continues on next page →

↪ *continued from previous page*

```
    x := (x + 4) +// 256                          ' move right, wrap
    waitms(20)
```

The sprite is defined once; the loop re-stamps it with one `SPRITE` per frame. For a shape built from many primitives the saving is larger still — the geometry is rasterized into the sprite once, and every later frame costs a single stamp.

The update model

The PLOT window has two update modes, chosen by whether you put `UPDATE` on the creation line.

Automatic mode is the default. Every drawing command repaints the window as it arrives, so the picture builds up live. This is the simplest mode and is right for a handful of primitives or a static figure.

Buffered mode is enabled by the `UPDATE` keyword on the creation line. In this mode your primitives accumulate on the off-screen canvas and **nothing is shown until you send a runtime `` `UPDATE `` command**. Buffered mode is the right choice for a scene built from many primitives, and it is essential for flicker-free animation: clear, redraw the whole frame, then `UPDATE` once.

```
PUB main() | f, ballx
  DEBUG(`PLOT Anim SIZE 512 256 BACKCOLOR $000000 UPDATE)   ' buffered
  ballx := 0
  repeat f from 0 to 200
    DEBUG(`Anim CLEAR)  ' erase (off-screen)
    DEBUG(`Anim COLOR $FFFF00 SET `(ballx) 128 CIRCLE 30 0 255)
    DEBUG(`Anim UPDATE)  ' present one frame
    ballx := (ballx + 4) +// 512
    waitms(20)
```

Three more commands round out display control:

- `` `CLEAR `` — fill the canvas with the background color and reset it for a new frame. In buffered mode this clears the off-screen canvas; the cleared state becomes visible at the next `UPDATE`.
- `` `SAVE `` — save the current canvas image to a BMP file on the host. Send `SAVE` for a default filename, or `SAVE 'name'` to choose one (the `.bmp` extension is added for you — do not include it).
- `` `CLOSE `` — close this window and free its resources.

`UPDATE` plays two roles. On the creation line it is the **flag** that turns buffered mode on. At runtime, `` `UPDATE `` is the **command** that presents the accumulated drawing. You enable buffering once with the creation-line `UPDATE`, then trigger each repaint with a runtime `` `UPDATE ``.

A complete worked example

This first program is a **tour of the drawing primitives** — it exercises the polyline, the scatter, text, and the coordinate system so you can see each one work, with no wiring and all of its own data generated on the P2. It plots one cycle of a sine wave from the CORDIC engine as a connected polyline, overlays a random scatter of dots from the hardware RNG, and labels the result. The origin is moved to the left-center so the wave sits around a center line — y already increases upward in the default coordinate system. The two worked instruments that follow put these same primitives to work on real tasks.

```
CON _clkfreq = 200_000_000

PUB main() | x, y, angle, i, sx, sy

  ' Create a 512x512 plotting canvas on a black background
  DEBUG(`PLOT Wave SIZE 512 512 BACKCOLOR $000000)

  ' Origin at left edge, vertical center; y increases upward (the default)
  DEBUG(`Wave ORIGIN 0 256)

  ' Center axis line in dim gray
  DEBUG(`Wave COLOR $404040)
  DEBUG(`Wave SET 0 0)
  DEBUG(`Wave LINE 511 0 1 255)

  ' One cycle of a CORDIC sine wave, drawn as a connected polyline.
  ' angle sweeps the full circle ($0000_0000..$FFFF_FFFF)
  ' across 512 columns.
  DEBUG(`Wave COLOR $00FF00)
  DEBUG(`Wave SET 0 0)
  repeat x from 0 to 511
    angle := x * ($FFFF_FFFF / 511)
    y := sine(angle, 200)                        ' amplitude 200 px
    DEBUG(`Wave LINE `(x) `(y) 1 255)

  ' Random red scatter using the hardware RNG
  DEBUG(`Wave COLOR $FF0000)
  repeat i from 0 to 99
    sx := rnd() +// 512                           ' 0..511
    sy := (rnd() +// 400) - 200                   ' -200..+199
    DEBUG(`Wave SET `(sx) `(sy))
```

continues on next page →

↪ *continued from previous page*

```
    DEBUG(`Wave DOT 4 200)

  ' White centered label
  DEBUG(`Wave COLOR $FFFFFF)
  DEBUG(`Wave SET 256 230)
  DEBUG(`Wave TEXT 16 'QSIN + scatter')

  repeat                                          ' keep the window open

PRI sine(angle, length) : result
  ORG
              SETQ      #0                         ' Y coordinate = 0
              QROTATE   length, angle  ' rotate (length, 0) by angle
              GETQY     result  ' result = length * sin(angle)
  end

PRI rnd() : r
  ORG
              GETRND    r  ' 32-bit hardware random value
  end
```

The `sine` helper drives the CORDIC solver directly: **QROTATE** rotates the point (`length`, 0) by `angle`, and **GETQY** returns `length x sin(angle)` — a software-only sine source that needs no lookup table and no hardware. `rnd` reads the on-chip random generator with **GETRND**. Both are wrapped in inline PASM so the example builds and runs on a bare P2 board.

A worked instrument: an analog gauge

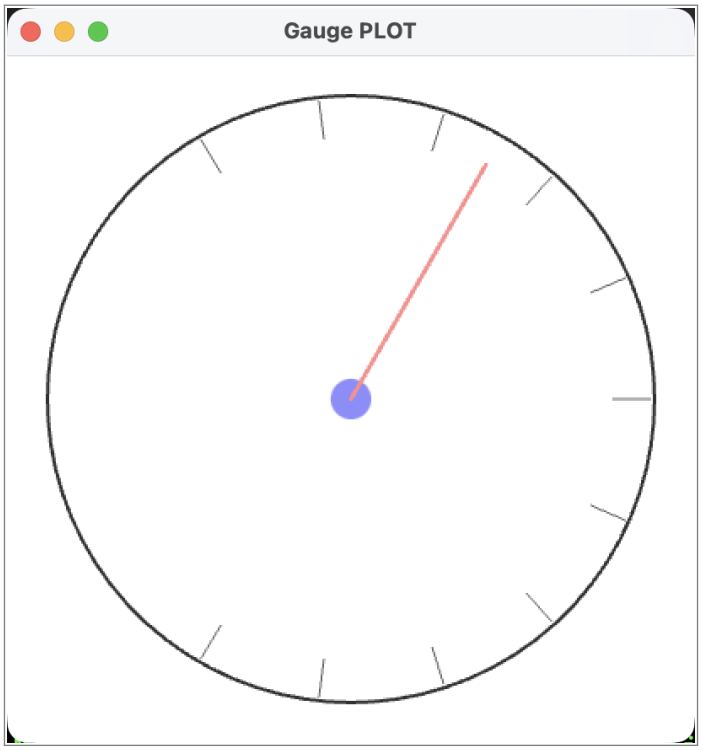

Figure 5.2. An analog gauge drawn in the PLOT window using polar coordinates.

Polar mode turns an instrument needle into a single `LINE`. Center the origin, switch to polar so the cursor's coordinates are (radius, angle), and the needle for any reading is one line from the center out to that reading's angle. The dial — tick marks and a ring — is drawn the same way, so a complete analog gauge needs only `LINE` and `CIRCLE`. This program sweeps a software-generated reading across a 240° scale, redrawing in buffered mode so it never flickers:

```
1  CON _clkfreq = 200_000_000
2
3  PUB main() | ang, value, needle, i, tick
4    ' Buffered gauge: redraw the whole dial + needle each frame, flicker-free.
5    DEBUG(`PLOT Gauge SIZE 400 400 BACKCOLOR $000000 UPDATE)
6    DEBUG(`Gauge ORIGIN 200 200)            ' (0,0) at the dial center
7    DEBUG(`Gauge POLAR 360)  ' angles in degrees; theta 0 points up
8
9    ang := 0
10   repeat
11     value  := 50 + qsin(50, ang, 360)  ' software-generated 0..100 reading
12     needle := (value * 240 / 100) - 120   ' map 0..100 -> -120..+120 degrees
```

continues on next page →

↪ continued from previous page

```
13
14      DEBUG(`Gauge CLEAR)
15
16      ' Scale: eleven radial ticks across the 240-degree sweep.
17      DEBUG(`Gauge COLOR $404040)
18      repeat i from 0 to 10
19        tick := (i * 24) - 120
20        DEBUG(`Gauge SET 90 `(tick) LINE 100 `(tick) 2 255)
21
22      ' Dial ring and center hub.
23      DEBUG(`Gauge SET 0 0 CIRCLE 210 2 255)
24      DEBUG(`Gauge COLOR $00FFFF SET 0 0 CIRCLE 12 0 255)
25
26      ' Needle: one polar line from center out to the reading's angle.
27      DEBUG(`Gauge COLOR $FF7F00 SET 0 0 LINE 95 `(needle) 4 255)
28
29      DEBUG(`Gauge UPDATE)                    ' present the frame
30
31      ang += 3
32      waitms(30)
```

Each frame clears the buffered canvas, draws the scale ticks and ring as polar lines and a centered circle, places the needle with one `LINE` at the reading's angle, and presents the frame with `UPDATE`. The reading here comes from `QSIN`; wire it to any value your program computes and the needle follows.

> Buffered mode (`UPDATE` on the creation line) keeps the gauge smooth: the whole dial is redrawn off-screen each frame, then shown at once. Without it you would see the needle erase-and-redraw flicker.

A worked instrument: a control-loop strip chart

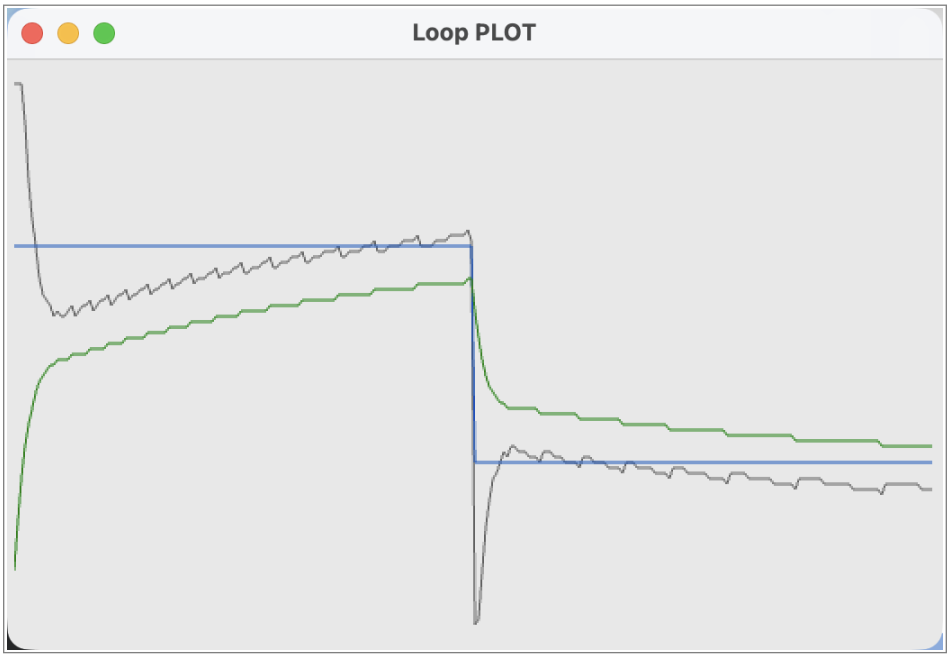

Figure 5.3. A PID control loop on the PLOT window: setpoint, the process variable's overshoot-and-settle, and the controller output.

A strip chart — several values traced against time — is how you watch a control loop behave while you tune it. This program runs a complete loop in software: a **PI controller** driving a simulated **first-order process**, with nothing wired up. The setpoint steps up and then back down; the process variable chases it, overshoots, and settles; the controller output is the effort that drives it there. Plotting all three together is exactly what you do on the bench to choose P, I, and D gains.

The loop is an honest little simulation. Each step computes the error, accumulates it for the integral term, forms the controller output (clamped, like a real actuator), and advances a first-order process model toward that output. The three histories are then drawn as three polylines across the canvas:

```
1  CON
2    _clkfreq = 200_000_000
3    STEPS = 256
4
5  VAR
6    LONG spH[STEPS], pvH[STEPS], ctlH[STEPS]
7
8  PUB main() | t, sp, pv, ctl, err, integ
9    DEBUG(`PLOT Loop SIZE 512 320 BACKCOLOR $E8E8E8)
10   DEBUG(`Loop ORIGIN 0 10)              ' baseline at bottom; y up (default)
```

continues on next page →

↪ *continued from previous page*

```
11
12    repeat                                ' re-run the experiment continuously
13      pv := 0
14      integ := 0
15      repeat t from 0 to STEPS-1
16        sp := (t < 128) ? 70 : 30         ' setpoint steps up then down
17        err := sp - pv
18        integ += err
19        ctl := (err*2 + integ/32) #> 0 <# 100   ' PI control, clamp 0..100
20        pv += (ctl - pv) / 10             ' first-order process lag
21        spH[t] := sp
22        pvH[t] := pv
23        ctlH[t] := ctl
24
25      DEBUG(`Loop CLEAR)
26      DEBUG(`Loop COLOR $606060)
27      trace(@ctlH)                        ' controller output (dark gray)
28      DEBUG(`Loop COLOR $0050C0)
29      trace(@spH)                         ' setpoint (blue)
30      DEBUG(`Loop COLOR $008000)
31      trace(@pvH)                         ' process variable (green)
32      waitms(500)
33
34  PRI trace(p) | t
35    DEBUG(`Loop SET 0 `(LONG[p][0] * 3))    ' value 0..100 -> 0..300 px
36    repeat t from 1 to STEPS-1
37      DEBUG(`Loop LINE `(t*2) `(LONG[p][t] * 3) 2 255)
```

`trace` draws one history array as a connected polyline: it `SET`s the cursor to the first point, then issues a `LINE` to each following point, scaling the 0–100 value range to the canvas height. Each pass re-runs the whole experiment and redraws all three traces together. Raise the proportional gain (the `*2`) or change the integral divisor (`/32`) and the overshoot and settling time shift exactly as a real loop's would — the chart *is* the tuning feedback.

Where you'd use this

In computer science and computer engineering, the PLOT window is the canvas for **control-systems work** — watching a loop respond and tuning it — and for general **instrumentation and data visualization**, where you build the exact gauge, chart, or figure your data calls for.

On an embedded project, you reach for it to plot a PID strip chart while tuning a motor or thermal loop (as here), to trace a battery's charge curve, to plot raw ADC against engineering units during a calibration, or to build a servo or RPM dial like the gauge above.

Bandwidth fit: these are all low-rate — readings and traces updating at tens to a few hundred points per second — and sit comfortably inside the link budget.

Extension (real hardware): replace the simulated process model with a real measured value (an ADC read, a sensor sample) and the simulated `ctl` with your actual control output, and the same strip chart shows a live loop.

Considerations

- **Match the coordinate system to the figure.** Set `ORIGIN` and the `CARTESIAN` flips (or `POLAR`) once at the top, in the coordinate convention your figure is natural in, and every primitive follows. A graph wants the origin at a corner with y up; a radial display wants the origin centered and polar mode.
- `linesize 0` **fills; greater than zero outlines.** For `CIRCLE`, `OVAL`, `BOX`, and `OBOX`, the linesize argument is what selects filled versus outlined. This is the most common source of an unexpectedly solid or hollow shape.
- `LINE` **moves the cursor;** `DOT` **and the shapes do not.** Chain `LINE` calls to draw a path; use `SET` before a `DOT`, `CIRCLE`, or `TEXT` to position it.
- **Use buffered mode for whole-scene redraws.** A static figure or a few primitives can run in automatic mode. For animation or a many-primitive scene, add `UPDATE` and present each frame with one runtime `` `UPDATE ``, which removes the flicker of per-primitive repainting.
- **Composite prebuilt layers and sprites instead of redrawing geometry.** When a background or a repeated element is fixed, load it once with `LAYER` (or define a sprite once with `SPRITEDEF`) and place it each frame with `CROP` or `SPRITE`. You re-issue one command rather than every primitive that produced the image. Remember that `LAYER` reads a `.bmp` from the host filesystem.
- **Layers are 1–8, sprite ids are 0–255.** Mixing these ranges up is a frequent error; the layer commands silently ignore an out-of-range index.

Try it

Start from the worked example. Then: switch the window to buffered mode by adding `UPDATE` to the creation line, wrap the whole drawing in a `repeat`, animate the phase by adding a growing offset to `angle`, and end each frame with `CLEAR` … draw … `` `UPDATE ``. You will have a scrolling sine wave with a fresh scatter each frame, built entirely from CORDIC and the RNG, using the coordinate system, primitives, color, and the buffered update model together.

Chapter 6: The LOGIC Window

Digital Waveforms

The LOGIC window is a digital-waveform visualizer. You send it sample values; it renders the individual bits of those samples as stacked logic traces, the way a hardware logic analyzer draws its channels. Each channel is one bit (or, in range mode, a small field of bits) of every sample, plotted left to right across a fixed time base.

The window shows raw logic levels and nothing more. It has **no protocol decoding** — there is no `DECODE` keyword and no built-in I2C, SPI, UART, CAN, or USB interpreter. If you want a byte value or a decoded transaction, your own Spin2/PASM2 code computes it; the LOGIC window only displays the lines. This is the single most important thing to keep in mind about the window, and this chapter returns to it in the worked example and the considerations.

You create one LOGIC window per `DEBUG(`LOGIC ...)` declaration, naming it and declaring its channels in that one statement, then feed it sample values by name.

> Keyboard and mouse input (`PC_KEY`, `PC_MOUSE`) work in the LOGIC window, but they share one mechanism across every window type and are covered in Chapter 12. Packed data formats are shared across the instrument windows and are detailed in Chapter 13; this chapter shows how LOGIC uses them.

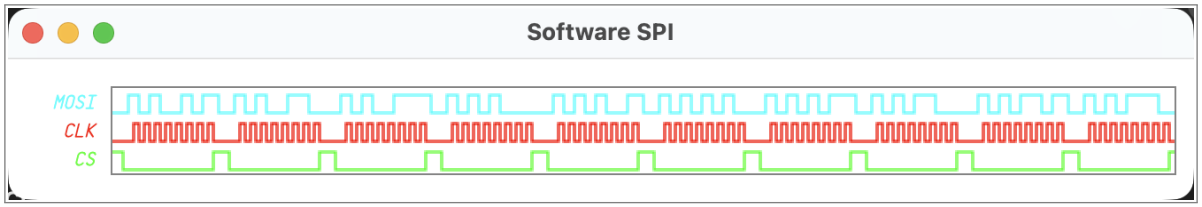

Figure 6.1. The LOGIC window showing eight channels of a binary ripple counter.

Creating a LOGIC window and declaring channels

You create the window, set its display options, and declare its channels in a single `DEBUG` statement. The first token after the backtick is the window type (`LOGIC`); the second is a name you choose. Channels are declared as **string and value elements inside the same statement** — a quoted label, optionally followed by a count, the `RANGE` keyword, and a color. There are no `CHANNELS`, `LABELS`, or `COLORS` keywords; the labels, counts, and colors *are* the channel declaration.

```
1 PUB main() | sample
2   ' create + declare 4 channels
```

continues on next page →

↪ *continued from previous page*

```
DEBUG(`LOGIC Bus SAMPLES 64 'CLK' $00FF00 'DATA' $FFFF00 'CS' 'WR')
DEBUG(`Bus `(sample))  ' feed it by name
```

That line creates a window named `Bus` with four single-bit channels: `CLK` (green), `DATA` (yellow), and `CS` and `WR` (default cycling colors). Channel 0 is the first declared label, channel 1 the second, and so on.

The configuration keywords you can add to the creation line:

Keyword	Arguments	Default	Range	What it sets
`TITLE`	`'text'`	`Logic`	—	The window's title-bar text
`POS`	`left top`	cascaded	screen px	Window position, in pixels
`SAMPLES`	`count`	`32`	4–2047	Horizontal resolution — samples shown across the width
`SPACING`	`pixels`	`8`	1–32	Horizontal pixels between samples
`RATE`	`divisor`	`1`	1–2048	Redraw once per `divisor` samples
`DOTSIZE`	`pixels`	`0`	0–32	Dot diameter at each sample (`0` = no dots)
`LINESIZE`	`pixels`	`3`	1–32	Waveform line thickness
`TEXTSIZE`	`points`	`10`	6–200	Channel-label font size
`COLOR`	`back grid`	black / gray	`$RRGGBB`	Background and grid colors
`HIDEXY`	—	shown	—	Hides the mouse-coordinate readout
`LONGS_1BIT` … `BYTES_4BIT`	—	unpacked	12 modes	Sets the data-packing mode (see "Packed sample data")

`SAMPLES` and `SPACING` together set the window width: width in pixels is `SAMPLES x SPACING`. The default `SAMPLES 32 SPACING 8` gives a 256-pixel-wide trace. Increasing `SAMPLES` shows more history; increasing `SPACING` stretches each sample wider.

> The window holds **up to 32 channels** and a single shared **2048-sample** circular buffer. The buffer is shared across all channels — every sample you send is one 32-bit value whose bits are distributed to the channels — not 2048 samples per channel. `SAMPLES` controls how many of those buffered samples are drawn, up to 2047.

Channel declaration syntax

Each channel element follows this form, with the optional parts in any of the combinations shown below:

> **Configuration Directive**
>
> `'label' {count} {RANGE} {color}`

- `'label'` — the channel name, drawn to the left of its trace.
- `count` — a number declaring multiple channels at once.
- `RANGE` — makes the channel a multi-bit value drawn as an analog-style waveform instead of a single high/low line.
- `color` — an `$RRGGBB` waveform color. Omit it and channels cycle through eight built-in colors (lime, red, cyan, yellow, magenta, blue, orange, olive).

One single-bit channel:

```
DEBUG(`LOGIC L 'CLK')              ' one channel, bit 0, default lime
```

Several single-bit channels from one label — a count after the label expands to that many consecutive single-bit channels, labeled `D 0`, `1`, `2`, …:

```
DEBUG(`LOGIC L 'D' 8)  ' 8 single-bit channels: D 0 .. D 7, bits 0..7
```

A multi-bit range channel — `RANGE` after a count makes a single channel that many bits wide, drawn as a waveform whose height tracks the value:

```
' one 8-bit channel, value 0..255 -> height
DEBUG(`LOGIC L 'ADC' 8 RANGE $FF0000)
```

A mix — declare them in the order you want them stacked (channel 0 at the bottom):

```
DEBUG(`LOGIC L 'CLK' 'COUNT' 8 RANGE $FF0000 'BUSY')
'                   bit0   bits 1..8 (8-bit range, red)  bit 9
```

If you declare no channels at all, the window defaults to all 32 channels, each a single bit, labeled `0`–`31`.

Sending sample data

Once the window exists, you feed it sample values by name. Each value you send is one **sample** — a 32-bit number whose bits are distributed across the declared channels: channel 0 takes the low bit (or low field, for a range channel), the next channel takes the bits above it, and so on. You send the *value* with `` `() ``:

```
DEBUG(`Bus `(sample))                ' one sample; its bits feed the channels
```

For the four-channel `Bus` above (`CLK DATA CS WR`, all single-bit), a sample value of `%1011` lights channel 0 (`CLK`) high, channel 1 (`DATA`) low, channel 2 (`CS`) high, channel 3 (`WR`) high. You build that value in your own code — from a counter, from `GETRND`, from port reads, or from a software-simulated signal — and the window draws whatever bits you send.

A range channel reads a *field* of bits rather than one bit. An 8-bit `RANGE` channel declared at the bottom of the stack reads bits 0–7 of each sample and maps that 0–255 value to the channel's vertical height, producing a stepped analog-style trace.

You can send several samples in one statement; each numeric element is one sample:

```
DEBUG(`Bus `(s0) `(s1) `(s2) `(s3))   ' four samples in one DEBUG call
```

Packed sample data

Sending one 32-bit long per sample is the simplest form, but it spends four serial bytes on every sample. When a sample needs only a few bits, a **packing mode** lets you carry many samples in one transmitted value. You set the mode as a keyword on the creation line; the window then unpacks each value you send into multiple samples.

Mode	Bits per sample	Samples per value
LONGS_1BIT	1	32
LONGS_2BIT	2	16
LONGS_4BIT	4	8
LONGS_8BIT	8	4
LONGS_16BIT	16	2
WORDS_1BIT	1	16
WORDS_2BIT	2	8
WORDS_4BIT	4	4
WORDS_8BIT	8	2
BYTES_1BIT	1	8
BYTES_2BIT	2	4
BYTES_4BIT	4	2

With LONGS_4BIT, for example, each long you send is unpacked into eight samples of four bits each — sample 0 in bits 0–3, sample 1 in bits 4–7, and so on:

```
PUB main() | packed, j
  DEBUG(`LOGIC Packed SAMPLES 256 LONGS_4BIT 'CLK' 'MOSI' 'MISO' 'CS')
  repeat
    packed := 0
    repeat j from 0 to 7                ' build 8 four-bit samples
      packed |= (GETRND() & $F) << (j << 2)
    DEBUG(`Packed `(packed))            ' one value -> eight samples
```

Packing trades serial bandwidth for the work of assembling the value in your code. The packing system is shared across the instrument windows and described in full in Chapter 13. The names are LONGS_, WORDS_, and BYTES_ followed by the bit width (_1BIT, _2BIT, _4BIT, _8BIT, _16BIT); there are no PACK1-style shortcuts and no run-length or compression modes.

Triggering

By default the window is **free-running**: every sample (after rate limiting) redraws the trace. To stabilize the display on a repeating event, set a trigger. The trigger watches a chosen set of channels for a specific bit pattern and aligns the display to the moment that pattern is matched.

Update Directive

```
TRIGGER mask match {offset}
```

- `mask` — which channels participate, one bit per channel. `$1` watches channel 0; `$3` watches channels 0 and 1; `0` disables the trigger (free-running).
- `match` — the bit values expected on the masked channels.
- `offset` — where the trigger event sits in the display, `0` to `SAMPLES-1`. `0` is the left edge (show what follows the event), `SAMPLES-1` is the right edge (show what led up to it), and the default is the center (`SAMPLES/2`).

The match **TEST** is `((sample XOR match) AND mask) = 0` — true when every masked bit equals its expected value. The trigger is **edge-sensitive**: it first has to see a sample that does *not* match (which arms it), then fires on the next sample that *does* match. This prevents a steady, already-matching signal from re-triggering on every sample.

```
' fire when channel 0 (CLK) goes high, event at sample 32
DEBUG(`Bus TRIGGER $1 $1 32)
' fire when ch0=1 and ch1=0, event 16 samples from left
DEBUG(`Bus TRIGGER $3 $1 16)
DEBUG(`Bus TRIGGER 0 0)            ' disable trigger (free-running)
```

You issue `TRIGGER` as a runtime command after the window exists; it is not a creation-line keyword. The trigger only evaluates once the displayed window of samples (the `SAMPLES` count) has filled.

Holdoff

After a trigger fires, `HOLDOFF` suppresses further triggers for a number of samples, so a busy or noisy signal does not re-trigger immediately:

Update Directive

```
HOLDOFF count
```

`count` ranges from 2 to 2048. After each trigger the window ignores trigger matches for `count` samples, then re-arms.

```
' after a trigger, skip 128 samples before re-arming
DEBUG(`Bus HOLDOFF 128)
```

Clearing and saving

Three runtime commands manage the display:

- `` `CLEAR `` — clears the trace, empties the sample buffer (`SamplePop` returns to zero), resets the trigger indicator, and resets the rate counter. The window starts collecting fresh samples from empty.
- `` `SAVE `` — saves the current window image to a `.bmp` file on the host.
- `` `CLOSE `` — closes this window and frees its resources.

```
DEBUG(`Bus CLEAR)                    ' empty the buffer and blank the trace
DEBUG(`Bus SAVE)  ' write the current image to a bitmap file
```

A complete software-only example

This program simulates an SPI master in Spin2 — no wiring, no probe — and shows its three lines (`CS`, `CLK`, `MOSI`) on the LOGIC window. Each call to `emit` packs the three logical levels into one sample (channel 0 = `CS`, channel 1 = `CLK`, channel 2 = `MOSI`) and sends it. The trigger aligns the display to the falling edge of `CS`, the start of each frame.

The decoding — knowing that this *is* SPI, that data is sampled on the rising clock edge, that the byte is `$A5` — lives entirely in this Spin2 code. The window shows only the three waveforms; it does not know they are SPI.

```
CON
  _clkfreq = 100_000_000

PUB main() | tx_byte, i, cs, clk, mosi
  DEBUG(`LOGIC SPIbus TITLE 'Software SPI' SAMPLES 200 SPACING 3 ...
         'CS' $00FFFF 'CLK' $00FF00 'MOSI' $FFFF00)
  ' align display to CS going low (frame start)
  DEBUG(`SPIbus TRIGGER $1 $0 32)

  tx_byte := $A5
  repeat
    ' idle: CS high, CLK low
    cs := 1
```

continues on next page →

↪ continued from previous page

```
    clk := 0
    mosi := 0
    emit(cs, clk, mosi)
    emit(cs, clk, mosi)

    ' start frame: assert CS low
    cs := 0
    emit(cs, clk, mosi)

    ' clock out 8 bits, MSB first
    ' (mode 0: data set on low, sampled on rising edge)
    repeat i from 7 to 0
      mosi := (tx_byte >> i) & 1
      clk := 0
      emit(cs, clk, mosi)                    ' present data with clock low
      clk := 1
      emit(cs, clk, mosi)                    ' rising edge

    ' end frame: release CS
    clk := 0
    cs := 1
    emit(cs, clk, mosi)

    tx_byte := (tx_byte + 1) & $FF           ' next byte

PRI emit(cs, clk, mosi) | s
  ' pack 3 lines into bits 0,1,2 of one sample
  s := cs | (clk << 1) | (mosi << 2)
  DEBUG(`SPIbus `(s))
```

Run it and the window shows `CS` framing each byte, eight clock pulses per frame, and `MOSI` carrying the bit pattern of `$A5`, `$A6`, `$A7`, … Because the trigger is set on `CS` low, every frame redraws in the same horizontal position and the display holds steady. To watch a multi-bit value instead, declare a `RANGE` channel and send the byte value directly:

```
DEBUG(`LOGIC Counter SAMPLES 256 'BYTE' 8 RANGE $00FF00)
repeat
  ' 8-bit ramp drawn as an analog-style trace
  DEBUG(`Counter `(value++ & $FF))
```

Acquisition: software-paced sampling and transition capture

The LOGIC window sits comfortably inside the link budget (Chapter 1), because digital debugging rarely needs a continuous full-rate stream. Two logic-specific habits keep it that way.

Software-paced — one sample per event. You do not have to sample a bus on every system-clock tick. The SPI example above sends one sample each time a line *changes* — idle, CS low, each clock edge — not one per clock cycle. Driving the window from your protocol's own events, rather than a free-running sample clock, is what keeps a bring-up trace small enough to stream live: a few hundred samples frame an entire transaction.

Transition + timestamp capture. When you do need to record a fast bus, store *transitions*, not samples. Each time a watched line changes, capture the new line state together with a timestamp (from `GETCT` or a free-running counter) and keep only those pairs. An idle bus then costs nothing while idle and a single entry per edge when it moves — far less than one sample per clock. The pairs pack tightly (Chapter 13), and you reconstruct the timing from the timestamps on the host. This is how a logic analyzer records minutes of a sparse bus without a giant buffer.

The mechanics underneath a fast capture — a circular buffer filled in a tight PASM loop, an arm/trigger/freeze cycle, dumping one frozen frame over the slow link — are shared with SCOPE; the acquisition section of Chapter 7 develops them once. LOGIC's trigger (above) arms and fires on a bit pattern the same way.

Where you'd use this

In computer science and computer engineering, the LOGIC window is the tool for **protocol engineering** — bringing up and verifying a serial bus — and for **debugging concurrent systems**, where what matters is the *timing relationship* between several digital signals.

On an embedded project, you reach for it during bit-banged-driver bring-up (does the clock idle in the right state, is data sampled on the correct edge — CPOL/CPHA), to check chip-select and bus-arbitration timing, to watch inter-cog signalling and lock hand-offs, and to confirm setup-and-hold against a datasheet's timing diagram.

Bandwidth fit: software-paced traces and buffered bursts stream comfortably; continuously monitoring a *fast* live bus does not **FIT**, and is the case the capture strategies above exist to handle.

Extension (real hardware): replace the simulated lines with real pin reads — sample the actual port bits into each sample value — and the same channels, trigger, and decode-in-code approach shows a live bus.

Considerations

- **The window shows waveforms; you decode in code.** LOGIC renders logic levels. Any interpretation — framing, byte values, protocol state — is computed by your Spin2 or PASM2 code before you send the sample. There is no decoder keyword. Treat the window as the lines on a logic analyzer's screen, not as a protocol analyzer.

- **One sample is one 32-bit value, distributed across channels.** Build the value so each channel's bit (or field) lands at the right offset: channel 0 at bit 0, the next channel at the bits above it. A range channel consumes a contiguous field.
- **The buffer is 2048 samples, shared, circular.** SAMPLES chooses how many are drawn (4–2047) — and only that many are ever marked valid, so the trigger evaluates and displays within the SAMPLES window, not the full 2048-deep buffer.
- **Trigger fires on an edge, not a level.** It must first see a non-matching sample, then a matching one. A signal already sitting at the match value will not trigger until it leaves and returns. Use HOLDOFF to keep a busy signal from re-triggering.
- **Pack when bandwidth matters.** One long per sample is simplest; the LONGS_/ WORDS_/BYTES_ packing modes carry many narrow samples per transmitted value at the cost of assembling that value in your code (Chapter 13).
- RATE **thins redraws, not samples.** A high RATE divisor reduces how often the trace repaints, lowering host load, while every sample still enters the buffer.
- **LOGIC vs. SCOPE.** Use LOGIC for discrete digital lines and bit patterns; use SCOPE (Chapter 7) for a continuously varying analog value over time. A RANGE channel bridges the two when you want a small multi-bit value shown as a stepped waveform alongside digital lines.

  See also: Chapter 7 (SCOPE) for analog time-domain traces, Chapter 12 for PC_KEY and PC_MOUSE input, and Chapter 13 for the shared packed-data formats.

Try it

Start with the SPI example above. Then: add a fourth channel MISO and have the simulated peripheral drive a reply byte back on it (compute the bits in code — the window still only shows the lines). Next, switch CS/CLK/MOSI/MISO to a packed feed using LONGS_4BIT, building each long from eight four-bit samples, and confirm the same waveforms appear with a quarter of the serial traffic. Finally, add an 8-bit RANGE channel that shows the byte value your code has assembled, so you can watch the decoded byte rise and fall beside the raw lines — proof that the decoding is yours and the display is just the waveforms.

Chapter 7: The SCOPE Window

Time-Domain Oscilloscope

The SCOPE window plots values against time, the way a bench oscilloscope does. You send it a stream of samples; it scrolls them across the display, newest at the right, oldest at the left. Where the TERM window shows you *what* a value is, the SCOPE window shows you *how it changes* — the shape of a waveform, the settling of a signal, the moment an event crosses a threshold.

One SCOPE window holds up to **eight channels**, each with its own label, color, vertical range, and position. Behind the display is a circular buffer of **2048 sample sets** per window — a set being one sample from every active channel. You declare the window once, naming its channels in the same statement, and then feed it bare numeric values for the rest of the run.

> SCOPE plots one value against time. For one value against *another* value — phase plots, Lissajous figures, XY trajectories — use the SCOPE_XY window in Chapter 8. Keyboard and mouse input (`PC_KEY`, `PC_MOUSE`) work here too and are covered for all windows together in Chapter 12. Packed data formats, which let you move samples faster over the debug link, are covered in Chapter 13.

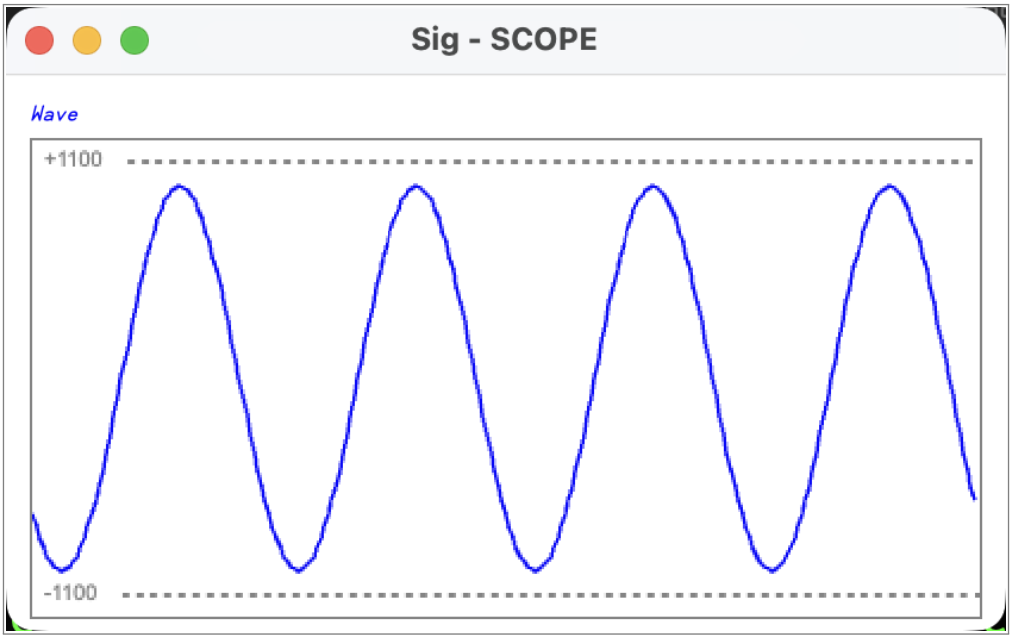

Figure 7.1. The SCOPE window displaying a time-domain sine waveform.

Creating the window and declaring its channels

You create and configure a SCOPE window in a single DEBUG statement. The first token after the backtick is the window type (SCOPE); the second is a name you choose and address the window by afterward. Configuration keywords and channel declarations follow on the same line:

```
PUB main() | ang
  ' create, one auto-ranging channel
  DEBUG(`SCOPE Sig SIZE 400 200 'Wave' AUTO)
  ang := 0
  repeat
    DEBUG(`Sig `(qsin(1000, ang, 256)))         ' feed it by name
    ang += 4
    waitms(5)
```

The configuration keywords you can place on the creation line:

Keyword	Arguments	Default	What it sets
TITLE	'text'	Scope	The window's title-bar text
POS	left top	cascaded	Screen position of the window, in pixels
SIZE	width height	256 256	Display size in pixels; each is **32–2048**
SAMPLES	count	256	Horizontal resolution — sets displayed at once; **16–2048**
RATE	divisor	1	Display-update divisor (see "Considerations"); **1–2048**
DOTSIZE	pixels	0	Dot diameter; **0–32** (0 = no dots)
LINESIZE	pixels	3	Line thickness; **0–32** (0 = no lines)
TEXTSIZE	points	10	Label font size; **6–200**
COLOR	back grid	black / gray	Background color, then grid color ($RRGGBB each)
HIDEXY	—	off	Hides the mouse-coordinate readout
Packing keyword	—	LONGS_1BIT	Sets the data-packing format (see Chapter 13)

If you set both DOTSIZE and LINESIZE to 0, the window forces a dot size of 1 so traces remain visible.

Channels are declared as elements, not keywords

You do not declare channels with a `CHANNELS` or `LABELS` keyword. Each channel is introduced by a **quoted label** in the creation stream, optionally followed by numeric arguments that configure that one channel:

Configuration Directive

```
'label' {AUTO | lo hi} {tall} {base} {grid} {color}
```

The window reads the label, then reads the optional numeric arguments **in order**:

Argument	Meaning	If omitted
`AUTO`	Auto-range this channel (mutually exclusive with `lo hi`)	manual range used
`lo hi`	Manual range: value at the bottom and top of the trace area	full 32-bit range
`tall`	Vertical span of the trace, in pixels	full window height
`base`	Vertical offset of the trace, in pixels	`0`
`grid`	Grid flags: 4-bit mask — bit0 baseline line, bit1 top line, bit2 min-value label, bit3 max-value label	`0` (off)
`color`	Trace color, `$RRGGBB`	next from the default palette

The first label after `SCOPE` becomes channel 0, the next becomes channel 1, and so on, up to eight. So this declares three channels:

```
DEBUG(`SCOPE Waves SIZE 512 300 SAMPLES 256 ...
  'Sine'  -1000 1000 100   0 0 $00FF00 ...
  'Tri'   -1000 1000 100 100 0 $FF0000 ...
  'Noise' -1000 1000 100 200 0 $00AAFF)
```

Each channel here has a fixed range of −1000 to 1000, is 100 pixels tall, and is offset vertically by `base` (0, 100, 200) so the three traces stack instead of overlapping. The `grid` argument is `0` (no grid lines or value labels), and the last value is the trace color.

> The arguments are positional. To set `color`, you must supply the arguments before it. If you want auto-ranging *and* a specific color, give `AUTO` followed by the `tall`, `base`, and `grid` values, then the color — for example `'Wave' AUTO 100 0 0 $00FF00`.

AUTO versus a manual range

A channel declared `AUTO` rescales itself on every redraw: the window scans the channel's samples currently in the buffer, finds their minimum and maximum, and maps that span to the trace height. This tracks an unknown signal without your having to know its amplitude in advance, at the cost of a display whose scale shifts as the signal changes.

A channel declared with `lo hi` keeps a fixed scale: `lo` sits at the bottom of the trace area and `hi` at the top, regardless of what the samples do. Use a manual range when you know the signal's bounds and want a stable display you can read against. If `lo` is greater than `hi`, the display is simply inverted — high values at the bottom.

Sending samples

Once the window exists, you feed it by name. Send **one bare numeric value per active channel**, in channel order, using the `` `() `` value form:

```
DEBUG(`Waves `(sine) `(tri) `(noise))   ' three channels: one set
```

Each complete set of values — one per channel — advances the time base by one column. The window stores the set in its circular buffer and scrolls the display. With a single channel you send a single value:

```
DEBUG(`Sig `(value))
```

The sample timing is entirely yours: the window plots a set whenever one arrives, so the spacing of your `DEBUG` calls in the loop is what determines the time scale. A `waitms` or `WAITX` in the loop sets how fast samples are produced.

> Send sample values with the `` `() `` form, which transmits the *value*. This is the same distinction as in the TERM chapter: `` `udec_(x) `` would send the visible *digits* of `x`, which is not what a sample stream wants.

Triggering

By default a SCOPE window free-runs: it redraws continuously as samples arrive. A **trigger** instead holds the display until the signal does something specific — crosses a level in a particular direction — so a repeating waveform appears stationary and a one-shot event is captured at a known position.

You configure the trigger at runtime with the `TRIGGER` command:

Update Directive

```
TRIGGER channel {AUTO | arm fire} {offset}
```

Argument	Meaning
`channel`	Which channel to watch: `-1` disables the trigger (free-run), `0`–`7` selects a channel
`AUTO`	Auto-trigger — the window computes the arm and fire levels from the signal's range
`arm fire`	Manual arm and fire levels (use these *instead of* `AUTO`)
`offset`	Where the trigger point sits in the display, `0..SAMPLES-1` (default: `SAMPLES/2`)

```
' channel 0, arm -500, fire 500, centered
DEBUG(`Capture TRIGGER 0 -500 500 256)
DEBUG(`Capture TRIGGER 0 AUTO)  ' channel 0, levels chosen automatically
DEBUG(`Capture TRIGGER -1)                ' disable: back to free-running
```

Direction is set by the levels, not a keyword

There is no `RISING` or `FALLING` keyword. The direction follows from how `fire` compares to `arm`:

- `fire` >= `arm` → **rising-edge trigger.** The window arms when the signal falls to or below `arm`, then fires when it rises to or above `fire`.
- `fire` < `arm` → **falling-edge trigger.** The window arms when the signal rises to or above `arm`, then fires when it falls to or below `fire`.

The two-level scheme (arm one place, fire at another) gives the trigger hysteresis, so noise around a single threshold does not produce repeated false triggers. To trigger on a rising signal at 500, set `arm` below it and `fire` at 500 (`TRIGGER 0 -500 500`); to trigger on a falling signal at 500, put `arm` above `fire` (`TRIGGER 0 700 500`).

With `AUTO`, the window scans the trigger channel's range and sets `arm = low + range/3` and `fire = low + range/2` — a rising-edge trigger near the middle of the signal.

Trigger position with offset

The `offset` argument places the trigger point within the displayed window, measured in samples from `0` to `SAMPLES-1`:

- `0` puts the trigger at the **right** edge — it tests the newest sample, so you see the lead-up *to* the event (pre-trigger history).
- `SAMPLES-1` puts it at the **left** edge — it tests the oldest sample, so you see what happens *after* the event (post-trigger).

- `SAMPLES/2` (the default) centers it, showing equal time before and after.

The trigger is evaluated only once the buffer holds a full `SAMPLES` worth of data, so the pre-trigger region is always populated.

HOLDOFF

After a trigger fires, `HOLDOFF` suppresses re-triggering for a number of samples, which steadies the display of a busy or bursty signal:

```
' ignore new triggers for 512 samples after one fires
DEBUG(`Capture HOLDOFF 512)
```

The holdoff count ranges from **2 to 2048**. It defaults to `SAMPLES` — one full screen — so by default the window will not re-trigger until it has shown the captured frame.

Clearing and saving

Three more runtime commands round out the set:

- `` `CLEAR `` — clears the display and resets the sample buffer, so the next samples start a fresh trace from the right edge.
- `` `SAVE `` — saves the current display image to a `.bmp` file on the host. An optional filename may follow; without one, the host names the file.
- `` `CLOSE `` — closes this window and frees its resources.

```
DEBUG(`Sig SAVE)    ' write the current trace to a bitmap on the PC
DEBUG(`Sig CLEAR)   ' wipe the buffer and start over
```

A complete worked example

This program needs no wiring. It generates three software waveforms and plots them on three stacked SCOPE channels: a CORDIC sine (`QSIN`), a counter-driven triangle, and random noise from `GETRND`. It compiles with `pnut_ts` and runs on a bare P2 board with `pnut_term_ts` open.

```
CON
  _clkfreq = 200_000_000

PUB main() | ang, sine, tri, dir, noise
  ' Three stacked channels: fixed -1000..1000 range,
  ' 100px tall, offset by 'base'
```

continues on next page →

↪ *continued from previous page*

```
  DEBUG(`SCOPE Waves SIZE 512 300 SAMPLES 256 LINESIZE 2 ...
    'Sine'  -1000 1000 100   0 0 $00FF00 ...
    'Tri'   -1000 1000 100 100 0 $FF0000 ...
    'Noise' -1000 1000 100 200 0 $00AAFF)

  ang := 0
  tri := -1000
  dir := 40

  repeat
    ' CORDIC sine, amplitude 1000, 256 steps/cycle
    sine  := qsin(1000, ang, 256)
    tri   += dir                            ' ramp up/down for a triangle
    if tri >= 1000 OR tri <= -1000
      dir := -dir
    noise := (GETRND() // 2001) - 1000      ' random in -1000..1000

    ' one set: three values, in channel order
    DEBUG(`Waves `(sine) `(tri) `(noise))

    ang += 4
    waitms(5)                               ' your loop sets the time scale
```

Acquisition: catching a fast event over a slow link

Chapter 1 named the debt plainly: every sample you send crosses the 2 Mbaud **DEBUG** link, so a *continuous* full-rate stream of a fast signal will not fit. The SCOPE window is still the right tool for fast signals — you just stop streaming them live and use the two techniques every bench oscilloscope uses instead. Both are fully synthetic here; neither needs hardware.

Capture and dump — full fidelity, one window per trigger

The **capture-and-dump** strategy (Chapter 1) decouples the measurement from the link. A tight loop — in PASM when you need full speed — fills a **circular buffer at the P2's own sample rate**, overwriting oldest with newest, while it tests a trigger condition on every pass: a level crossing, a bus pattern, an out-of-range fault. When the trigger fires, you **freeze the buffer and dump it once** over the link — the pre-trigger samples already captured plus a post-trigger tail. The fidelity of what you caught is set by **how fast your loop runs and how deep your buffer is, not by the link**, which only carries the one readout. You see one frame per trigger and are blind between events; that is the price of full detail, and it is exactly the oscilloscope's *arm* → *acquire* → *trigger* → *freeze* → *read out* model.

The SCOPE window models this directly: a `TRIGGER` holds the display on the event and freezes the frame. Recasting the free-running display as a **one-shot capture** is one line of setup. Here the window waits for the signal to rise past 500 (armed below at −500, fired at or above 500 — `fire` >= `arm`, so rising) and freezes a 512-sample frame with the trigger point centered:

```
CON
  _clkfreq = 200_000_000

PUB main() | ang, sig
  ' SCOPE takes its channel declaration as a feed to the window name
  DEBUG(`SCOPE Capture SIZE 512 256 SAMPLES 512)
  DEBUG(`Capture 'Signal' -1000 1000)
  ' rising edge, trigger centered in the frame
  DEBUG(`Capture TRIGGER 0 -500 500 256)
  DEBUG(`Capture HOLDOFF 512)  ' one frame of holdoff before re-arming

  ang := 0
  repeat
    sig := qsin(1000, ang, 256)
    DEBUG(`Capture `(sig))
    ang += 3
    waitms(2)
```

The trigger and the buffer belong to the acquisition, not to the link. In a real high-rate capture you would move that arm-and-test loop into PASM so it runs at full speed, and only the frozen frame would ever cross the wire.

Worked example: catch a rare glitch

This is the capture-and-dump pattern end to end, fully synthetic. A clean, low-amplitude signal — it stands in for any quiet line you are watching — occasionally throws a single out-of-range spike. A `TRIGGER` set well above the clean signal's range arms on the quiet signal and fires the moment a spike crosses it, freezing a 512-sample frame with the glitch centered, its lead-up to the left and its aftermath to the right:

```
CON
  _clkfreq = 200_000_000

PUB main() | ang, n, sig
  DEBUG(`SCOPE Glitch SIZE 512 256 SAMPLES 512)
  DEBUG(`Glitch 'Signal' -1000 1000)
```

continues on next page →

↪ continued from previous page

```
' arm on the quiet signal, fire on the spike (fire >= arm -> rising)
DEBUG(`Glitch TRIGGER 0 200 800 256)   ' centered: see before and after
DEBUG(`Glitch HOLDOFF 512)

ang := 0
n   := 0
repeat
  sig := qsin(300, ang, 256)           ' clean signal, well inside range
  if n // 233 == 0                     ' a rare, occasional fault
    sig := 950                         ' one out-of-range sample
  DEBUG(`Glitch `(sig))
  ang += 5
  n   += 1
  waitms(1)
```

The display free-runs on the quiet signal until a spike crosses 800; then it locks, showing the glitch at the center of the frame with the clean signal before and after it. `HOLDOFF` keeps it from re-arming until you have had a full frame to read the captured event.

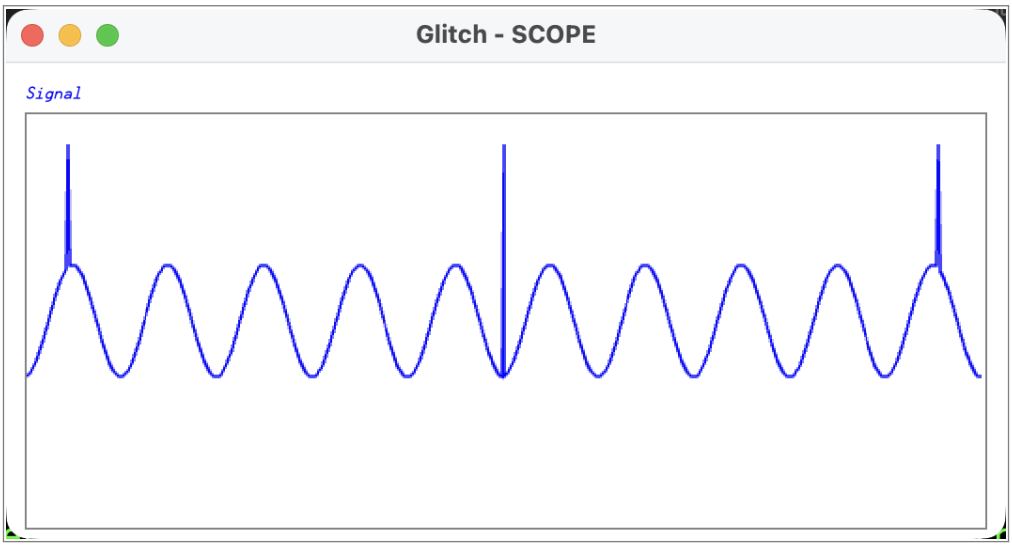

Figure 7.2. A rare out-of-range glitch frozen by a SCOPE trigger, shown with its lead-up and aftermath.

Decimate — a live trend, at the cost of detail

When you want a *continuous* view of a slowly-evolving signal rather than a frozen event, **decimate** (Chapter 1): send only one sample in every N. The window updates forever; you have traded resolution for a live trend. The naive form — take every Nth sample — carries a trap any instrument engineer will warn you about: a narrow spike that lands between the kept samples simply disappears, and a periodic signal can alias into a slower phantom.

```
' naive: keep 1 in 8 -- a one-cycle spike between samples is lost
if n // 8 == 0
  DEBUG(`Trend `(sig))
```

The honest fix is **min/max (peak) decimation**: over each group of N samples keep both the smallest and the largest, and send both. A one-cycle spike now survives, because it lands as the group's min or max even though you never sent that exact sample:

```
' peak decimation: keep the extremes of each group of 8
lo := lo <# sig                          ' running min (<# = limit max)
hi := hi #> sig                          ' running max (#> = limit min)
if n // 8 == 7
  DEBUG(`Trend `(lo) `(hi))              ' two traces preserve the spike
  lo := POSX                             ' reset for the next group
  hi := NEGX
```

Decimation is always-on but lossy; capture-and-dump is perfect but episodic. **Choose by whether you are watching a *trend* or hunting an *event*.**

Where you'd use this

In computer science and computer engineering, the SCOPE window is the everyday tool for **DSP work** — inspecting a waveform, the response of a filter, the settling of a control loop — and for **power and control electronics**, where the shape of a signal in time is the measurement.

On an embedded project, you reach for it to watch an ADC capture, a PWM edge or duty cycle, supply ripple or inrush at power-on (a triggered one-shot), contact bounce on a switch, or an intermittent fault line — the glitch-capture pattern above.

Bandwidth fit: low-rate live signals stream comfortably; a fast transient is caught as a triggered one-shot; a *continuous* high-rate analog stream does not **FIT**, and is the case the acquisition strategies above exist to handle.

Extension (real hardware): replace the synthetic `qsin`/spike source with a real sampled input — read an ADC or a smart pin in the loop — and the same channel, trigger, and capture code shows the live signal.

Considerations

- `RATE` **is a display-update divisor, not a sample rate in Hz.** With `RATE 1` (the default) the window redraws on every sample set. With `RATE 16` it accepts and buffers every set but redraws only on every sixteenth — which lowers the host's drawing load for fast streams. It does not change how often *you* send samples.

- **You set the time scale, not the window.** The horizontal axis is one column per sample set. How fast time appears to move is set by the spacing of your `DEBUG` calls — the `waitms`/`WAITX` in your loop — not by any window parameter.
- `AUTO` **adapts; a manual range stays put.** Auto-ranging follows an unknown signal but shifts scale as the signal changes; a fixed `lo hi` gives a stable, readable display when you know the bounds. For a steady trace, prefer a manual range.
- **Stack channels with** `base`**, scale them with** `tall`**.** Give each channel a height (`tall`) smaller than the window and a stepped `base` offset to lay multiple traces out without overlap, as the worked example does.
- `SAMPLES` **sets horizontal resolution and trigger depth.** It is also the default holdoff and the default trigger offset, so raising `SAMPLES` widens the captured frame and the pre-trigger window together.
- **For high sample rates, pack the data.** Bare per-channel values are simplest; the packing keywords (Chapter 13) move more samples per `DEBUG` packet over the link.

Try it

Start from the three-channel example. First switch the `Sine` channel from its fixed `-1000 1000` range to `AUTO` and watch the trace rescale on its own as you change the amplitude argument to `qsin`. Then add a trigger on the sine channel (`DEBUG`(Waves TRIGGER 0 -500 500 256)) `AND observe the waveform stand still instead of scrolling. Finally, vary the trigger`offset`between`0,SAMPLES/2, `AND`SAMPLES-1‘ to move the trigger point from the right edge to the center to the left edge, and see the pre-trigger region grow.

Chapter 8: The SCOPE_XY Window

XY, Lissajous, and Phase Plots

The SCOPE_XY window plots one value against another. Where the SCOPE window (Chapter 7) shows a signal *over time* — amplitude on the vertical axis, time marching across the horizontal — SCOPE_XY puts the *first* value on the X axis and the *second* on the Y axis, and draws a dot where they meet. Feed it a stream of `(x, y)` pairs and it draws the path those pairs trace out: a Lissajous figure from two oscillators, the phase relationship between two signals, the orbit of a moving point, a polar curve.

Reach for SCOPE_XY when the relationship between two values matters more than how either one changes over time. Two sine sources at the same frequency draw an ellipse whose shape encodes their phase difference; the same two at a 3:2 frequency ratio draw a stable knot. A position `(x, y)` plotted continuously becomes a trajectory. SCOPE is the right tool for a waveform you want to read left to right; SCOPE_XY is the right tool when the *shape* in the plane is the thing you want to see.

> Keyboard and mouse input (`PC_KEY`, `PC_MOUSE`) work in SCOPE_XY as in every window; they share one mechanism documented in Chapter 12. This chapter is about output — configuring the plot and feeding it coordinate pairs.

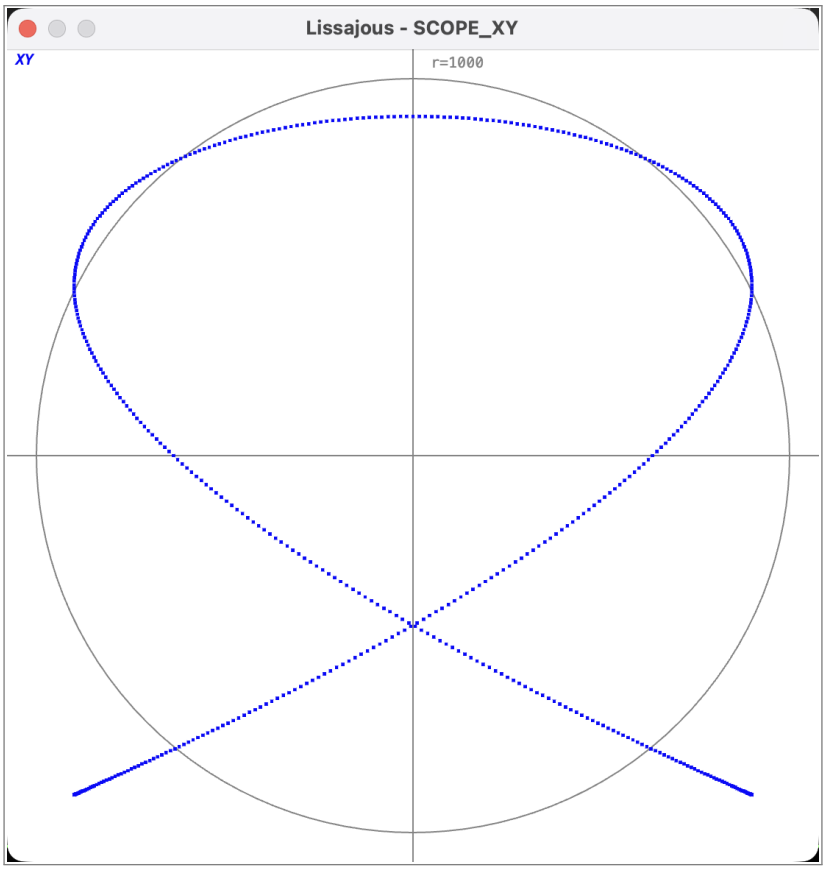

Figure 8.1. The SCOPE_XY window tracing a 2:3 Lissajous figure.

Creating a SCOPE_XY window

You create and configure the window in one `DEBUG` statement. The first token after the backtick is the window type (`SCOPE_XY`); the second is a name you choose. Channel names — the strings that label each trace — go on the same line, and each may be followed by a color:

```
PUB main()
  ' create + name
  DEBUG(`SCOPE_XY Lissajous SIZE 256 RANGE 1000 SAMPLES 0 'XY')
  DEBUG(`Lissajous `(500, 250))  ' feed by name
```

The configuration keywords you can add to the creation line:

Keyword	Arguments	Default	What it sets
TITLE	'text'	- SCOPE_XY	The window's title-bar caption (with no TITLE, the caption is <name> - SCOPE_XY)
POS	left top	cascaded	Screen position of the window, in pixels
SIZE	radius	256×256	Display **radius** in pixels; the plot is 2*radius wide and tall, and always square. With no SIZE, the plot is 256×256 (the default is a 256-pixel width, not a radius)
RANGE	value	$7FFFFFFF	Symmetric coordinate extent: the plot spans -value to +value on both axes (in polar mode, 0 to value for the radius)
SAMPLES	count	256	Persistence depth: how many recent points are kept and faded. 0 means infinite persistence — points accumulate and never fade
RATE	divisor	1	Plot one display update per this many samples received
DOTSIZE	pixels	6	Dot diameter, 2–20
TEXTSIZE	points	10	Legend text size, 6–200
COLOR	back {grid}	black, gray	Background color and, optionally, grid color
POLAR	{twopi {offset}}	—	Interpret pairs as (radius, angle) instead of (x, y) — see below
LOGSCALE	—	linear	Logarithmic radial scale, to magnify points near the center
HIDEXY	—	shown	Hide the X,Y coordinate readout at the mouse pointer
'name' {color}	—	next default color	Declare a channel (trace), optionally with a color

Three of these behave differently from what their names might suggest, and getting them wrong is the most common SCOPE_XY mistake:

- SIZE **takes one number, a radius — not a width and a height.** SIZE 256 produces a 512×512 plot. The window is always square; you cannot give it separate width and height.
- RANGE **takes one number, a symmetric extent — not four edges.** RANGE 1000 maps -1000 to the left/bottom edge and +1000 to the right/top edge. The origin is the center of the plot. A point at (0, 0) lands dead center; a point at (1000, 1000) lands in the top-right corner.
- SAMPLES **sets persistence depth, not a sample count to capture.** With SAMPLES 0 every point you send stays on screen forever (until you clear it) — use this for figures you want

to build up, like a complete Lissajous curve. With `SAMPLES 60` only the most recent 60 points are kept, and they fade from fully opaque (newest) to nearly transparent (oldest), drawing a comet-like trail behind a moving point.

Declaring channels

Each quoted string on the creation line declares one trace and names it for the legend. SCOPE_XY supports up to **8** traces. Put a color after a name to set that trace's color; omit it to take the next default:

```
DEBUG(`SCOPE_XY Phase SIZE 256 RANGE 1000 SAMPLES 200 'A' RED 'B' GREEN)
```

That declares two traces — `A` in red, `B` in green. The number of channels you declare determines how SCOPE_XY groups the numbers you feed it, which is the next section.

Sending coordinate pairs

Once the window exists, you feed it by name. Each pair of numbers you send is one `(x, y)` point. Send the values inside a `` `() `` group, comma-separated:

```
DEBUG(`Lissajous `(x, y))
```

This is the same `` `() `` value syntax used throughout the DEBUG windows: it sends the *values* of `x` and `y` as data, not their decimal text. There is no separate "plot a point" keyword — a pair of numbers *is* a point.

With more than one channel declared, send all channels' coordinates in one feed, in channel order — channel 0's X and Y first, then channel 1's, and so on:

```
' two channels declared as 'A' and 'B':
DEBUG(`Phase `(x1, y1, x2, y2))   ' (x1,y1) -> A, (x2,y2) -> B
```

SCOPE_XY collects values until it has a complete set — two per declared channel — then plots all channels at once and starts the next set. You can also split a set across several feeds; the window assembles them in arrival order.

Polar mode

Add `POLAR` to the creation line and SCOPE_XY interprets each pair as `(radius, angle)` instead of `(x, y)`. The first value is the distance from center (`0` to `RANGE`); the second is an angle.

```
DEBUG(`SCOPE_XY Rose SIZE 256 RANGE 1000 POLAR 360 'Rose')
```

`POLAR` takes up to two optional numbers:

- `twopi` — the angle value that equals one full circle. `POLAR 360` means angles are in degrees; `POLAR 1000` means a full turn is 1000 units. The default full-circle value is `$1_0000_0000` (a full 32-bit angle); pass `0` to select that default explicitly, or `-1` to run angles the other direction.
- `offset` — an angular offset added to every angle, rotating the whole plot.

Angle `0` points up; increasing angle sweeps around the circle. Feed `(radius, angle)` pairs exactly as you feed `(x, y)` pairs in Cartesian mode:

```
DEBUG(`Rose `(radius, angle))
```

LOGSCALE

`LOGSCALE` switches the radial axis from linear to logarithmic, magnifying points near the center so a wide range of magnitudes is visible at once. It applies in both Cartesian and polar modes — in Cartesian mode each point's distance from the origin is scaled logarithmically while its direction is preserved. Use it when your data spans several orders of magnitude and the small values would otherwise crowd into a dot at the center.

Clearing and saving

Two runtime commands you send by the window's name:

- `` `CLEAR `` — clears the plot and empties the sample buffer, then waits for new data. Use it to start a fresh figure, especially in persistent mode (`SAMPLES 0`) where points otherwise never disappear.
- `` `SAVE 'filename' `` — writes a `.bmp` image of the plot area to the host (the `.bmp` extension is appended automatically — give the name without it). Add the keyword `WINDOW` before the filename to capture the entire window instead of just the plot.

```
DEBUG(`Lissajous CLEAR)              ' wipe the plot
DEBUG(`Lissajous SAVE 'figure')      ' save the plot area to figure.bmp
```

A third runtime command, `` `CLOSE ``, closes the window.

A complete example: a Lissajous figure

This program drives one SCOPE_XY window with two software sine sources — no external hardware. The X coordinate is a sine of the phase; the Y coordinate is a sine of three times the phase. Two sines at a 3:1 frequency ratio draw a stable Lissajous figure. Both come from the P2's CORDIC engine via the **QSIN** method, whose signature is `QSIN(length, step, stepsInCircle)` — it returns `length x sin(step / stepsInCircle x 2pi)`. Passing `360` for `stepsInCircle` lets you treat `step` as degrees.

```
CON _clkfreq = 100_000_000

PUB main() | ph, x, y
  ' SIZE 256 -> 512x512 plot; RANGE 1000 -> axes span -1000..+1000;
  ' SAMPLES 0 -> persistent (the whole figure accumulates)
  DEBUG(`SCOPE_XY Lissajous SIZE 256 RANGE 1000 SAMPLES 0 'XY')
  ph := 0
  repeat
    x := QSIN(1000, ph, 360)        ' amplitude 1000, fills the range
    y := QSIN(1000, ph * 3, 360)    ' three times the X frequency
    DEBUG(`Lissajous `(x, y))       ' one (x, y) point per pass
    ph += 1
    waitms(5)
```

`QSIN` scales its output by the `length` argument, so an amplitude of `1000` matches the `RANGE 1000` you set — the figure fills the plot without clipping. The `waitms(5)` paces the points so you can watch the curve draw itself; remove it and the figure appears at once.

To turn this into a moving point with a trail instead of a static figure, change `SAMPLES 0` to a positive depth — say `SAMPLES 60` — and the window keeps only the 60 newest points, fading the older **ONES**. With a single rotating vector that produces a comet sweeping around a circle:

```
    x := QCOS(800, ph, 360)         ' QCOS gives the X leg
    y := QSIN(800, ph, 360)         ' QSIN gives the Y leg -> a circle
```

Where you'd use this

In computer science and computer engineering, SCOPE_XY is the tool for **I/Q constellations** — the QAM and PSK symbol clouds of digital communications — and for **phase-plane and dynamical-systems work**, where a system's state is read from the shape its two variables trace.

On an embedded project, you reach for it to plot a quadrature encoder's A and B channels as an XY circle (smart-pin captured), to watch a motor's d–q / FOC vector, to see a PLL pull an ellipse into a line as it locks, or to read an accelerometer's X–Y tilt as a moving point.

Bandwidth fit: decimated or low-rate pairs draw cleanly; a full-rate I/Q stream does not fit the link and is out — sample down to a rate the link carries.

Extension (real hardware): feed real `(x, y)` pairs — two ADC channels, the encoder's A/B counts, the accelerometer's two axes — in place of the synthetic sines, and the figure shows the live relationship.

Considerations

- `SIZE` **is a radius;** `RANGE` **is a symmetric extent.** A `SIZE 256 RANGE 1000` plot is 512×512 pixels and maps data from `-1000` to `+1000` on each axis. Match your data's amplitude to `RANGE` so the figure fills the plot without clipping; values beyond `+/-RANGE` fall outside the visible area.
- **Choose persistence to the job.** `SAMPLES 0` builds a complete, permanent figure — right for Lissajous curves and phase portraits you want to read whole. A positive `SAMPLES` value draws a fading trail — right for a moving point whose recent path matters more than its full history. Larger depths cost more to redraw each update, since every kept point is re-plotted with its faded opacity.
- **Use** `RATE` **to throttle fast data.** With `RATE n` the window repaints once per `n` samples received. This reduces host load when you are feeding points faster than you need to see them; the samples between repaints are still buffered.
- **Send pairs, not formatted text.** `` `(x, y) `` sends the *values* as a coordinate pair. A backtick formatter like `` `udec_(x) `` would send the decimal digits as a label string, not plot a point — the same value/format distinction as every other window.
- **Match amplitude to** `RANGE` **with QSIN's** `length`**.** Because `QSIN`/`QCOS` scale by their `length` argument, setting `length` equal to `RANGE` makes a unit-circle signal fill the plot. Scale `length` down to shrink the figure.
- **SCOPE vs. SCOPE_XY.** Use SCOPE (Chapter 7) for a value over time; use SCOPE_XY for one value against another. Two signals you would view as separate traces in SCOPE become a single shape in SCOPE_XY, and that shape is what reveals their phase and frequency relationship.

Try it

Start from the Lissajous example. Then change the frequency ratio — make `y` use `ph * 2` instead of `ph * 3`, and watch the figure change shape; a `2:1` ratio draws a different knot than `3:1`. Next, add a small offset to the Y phase (`QSIN(1000, ph * 3 + 30, 360)`) and watch the figure rotate and open — that offset is exactly the phase difference a real pair of signals would show. Finally, switch `SAMPLES 0` to `SAMPLES 80` to trade the static figure for a moving, fading trace, and add a second channel (declare `'XY'` and a second `'Orbit'` with its own color, then feed four numbers per pass) to see two figures share the plot at once. You will have used the creation config, coordinate-pair feeding, persistence, and multi-channel layout together.

Chapter 9: The FFT Window

Frequency Spectrum

The FFT window shows you the *frequency content* of a signal. Where SCOPE plots a value against time — the waveform itself — FFT takes a block of samples, transforms it, and plots magnitude against frequency. A steady tone that looks like a sine wave on SCOPE shows up on FFT as a single tall spike at that tone's frequency. Mix three tones together and you see three spikes; add noise and you see a low carpet under them.

The window runs a Cooley-Tukey FFT on the samples you feed it and displays the resulting **magnitude spectrum** — one point per frequency bin, drawn as a line, dot, or filled-bar trace. It supports up to **8 channels**, each transformed and drawn independently.

You create one FFT window per `DEBUG(`FFT ...)` declaration, name it, and feed it samples by that name. This chapter covers creating the window, setting the FFT size, feeding samples across one or more channels, the two amplitude controls (magnitude shift and log scale), reading the spectrum, and the runtime commands.

> Keyboard and mouse input (`PC_KEY`, `PC_MOUSE`) work in the FFT window, but they share one mechanism across every window type, so they are covered together in Chapter 12. This chapter is about the spectrum display.

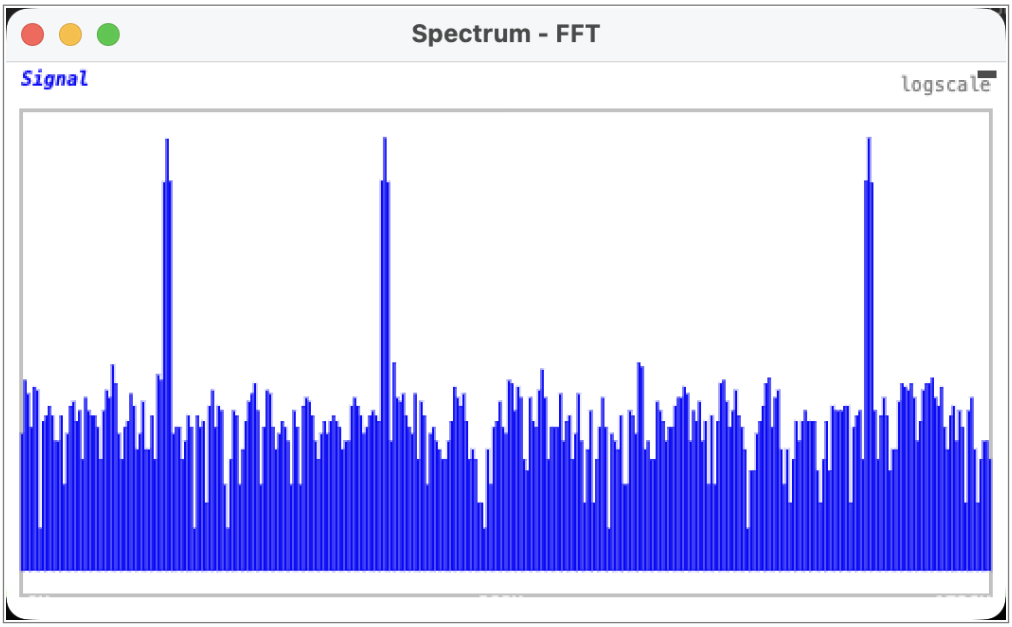

Figure 9.1. The FFT window showing two tones as spectral peaks.

What the FFT window does with your samples

You send a continuous stream of time-domain samples. The window stores them in a circular buffer and, once it has collected a full block of `N` samples, it:

1. Multiplies the block by a **Hanning window** to reduce spectral leakage.
2. Runs the FFT.
3. Computes the magnitude of each of the `N/2` frequency bins.
4. Draws the bins you asked for across the plot area.

The Hanning window is **applied internally on every transform** and is not configurable — there is no keyword to change it, and no choice of Hamming, Blackman, Flattop, or rectangular windowing. Every spectrum you see has been Hanning-windowed.

The transform produces `N/2` bins because a real-valued input signal has a spectrum that is symmetric about the Nyquist frequency; only the lower half carries unique information. Bin 0 is the DC (zero-frequency) component and bin `N/2 - 1` is the highest frequency the transform resolves.

Creating an FFT window

You create and configure the window in a single `DEBUG` statement. The first token after the backtick is the window type (`FFT`); the second is a name you choose. You feed it afterward by that name:

```
1 PUB main() | s
2   DEBUG(`FFT Spectrum SIZE 512 256 SAMPLES 1024)   ' create the window
3   repeat
4     repeat 1024
5       s := qsin(20000, GETCT(), $1_0000)           ' a sample
6       DEBUG(`Spectrum `(s))                         ' feed it by name
```

The configuration keywords you can add to the creation line:

Keyword	Arguments	Default	What it sets
TITLE	'text'	FFT	The window's title-bar text
POS	left top	auto	Screen position of the window, in pixels
SIZE	width height	—	Plot area in pixels; each is **32–2048**
SAMPLES	N {first last}	512	FFT size, and an optional displayed bin range
RATE	count	one per buffer	Redraw every count samples (**1–2048**)
DOTSIZE	radius	0	Dot radius in pixels (**0–32**)
LINESIZE	width	3	Line width (**−32–32**; negative draws filled bars)
TEXTSIZE	points	font (11)	Label font size; defaults to the window font (**6–200**)
COLOR	back grid	black/grey	Background color, then grid/frame color ($RRGGBB)
LOGSCALE	—	off	Logarithmic (log2-based) amplitude scaling
HIDEXY	—	off	Hides the coordinate readout
packing	—	off	Sample packing format (see "Packing samples")

SAMPLES is the one that defines the transform. The other essential is SIZE, which sets the pixel dimensions of the plot area.

Setting the FFT size with SAMPLES

SAMPLES N sets the FFT size. N must be a **power of 2 between 4 and 2048**; the value you give is clamped to that range and rounded down to the nearest power of 2, so SAMPLES 1000 becomes 512 and SAMPLES 1024 stays 1024. The default, if you omit SAMPLES entirely, is 512.

The window displays bins 0 through N/2 - 1 by default — the full spectrum. You can restrict the display to a contiguous **range of bins** by adding two more numbers:

```
DEBUG(`FFT Zoom SIZE 512 256 SAMPLES 1024 100 400)
```

This still runs a 1024-point transform but draws only bins 100 through 400, stretched across the full plot width — a zoom into one frequency region. The first bin must be in 0 ... N/2 - 2 and the last in first+1 ... N/2 - 1.

Feeding samples and declaring channels

Once the window exists, every bare number you send by its name is a **sample**. Samples are signed integers; the window collects them into its buffer and transforms a block at a time.

You declare a channel by sending a **string** — the channel's label — optionally followed by per-channel settings. The first string declares channel 0, the second declares channel 1, and so on, up to 8 channels. After a channel is declared, the samples you send are distributed across the declared channels in order: with two channels declared, samples alternate channel 0, channel 1, channel 0, channel 1.

A channel declaration takes these arguments, in order, all optional after the label:

Position	Meaning	Range
label	Channel name (string)	—
`MAG` shift	Magnitude bit-shift	**0–11**
high	Full-scale value for the Y axis	`1 ... $7FFF_FFFF`
tall	Channel height in pixels	—
base	Baseline offset from the bottom, in pixels	—
grid	Grid flags: bit 0 = baseline line, bit 1 = top line	—
color	Trace color (`$RRGGBB`)	—

A single green channel, full height, with a baseline grid line:

```
DEBUG(`Spectrum 'Signal' 0 $7FFF_FFFF 256 0 1 $00FF00)
```

Read that as: label `Signal`, magnitude shift 0, full scale `$7FFF_FFFF`, 256 pixels tall, baseline at the bottom (`base` 0), grid flag 1 (baseline line), color green.

To stack two channels in one window — say a left and right pair, one in the lower half and one in the upper half — declare both, then interleave their samples:

```
PUB main() | a, b
  DEBUG(`FFT Dual SIZE 512 256 SAMPLES 512)
  DEBUG(`Dual 'Left'  0 $7FFF_FFFF 128 0   1 $00FF00 ...
         'Right' 0 $7FFF_FFFF 128 128 1 $FF7F00)
  repeat
    repeat 512
      a := qsin(20000, GETCT(), $1_0000)
      b := qsin(10000, GETCT(), $1_0000)
      DEBUG(`Dual `(a) `(b))                ' one sample per channel
```

`Left` sits in the bottom 128 pixels (base 0), `Right` in the top 128 (base 128). Each channel is transformed independently and drawn in its own color. Channels are drawn back-to-front, so the first-declared channel ends up on top where they overlap.

Amplitude: magnitude shift and log scale

The FFT output is in arbitrary units — not decibels, and not an absolute scale. You have two independent controls over how tall the spectrum is drawn.

`MAG` **(per channel, 0–11)** is a bit-shift applied to the transform output: a `MAG` of `n` multiplies the magnitude by 2 . Use it to bring up a weak signal (`MAG 3` multiplies by 8) or to pull down one that saturates the top of the plot (`MAG 0`). It is set in the channel declaration, in the `MAG` shift position shown above.

`LOGSCALE` is a bare flag on the creation line. It applies a **log2-based** compression to the amplitude before drawing, expanding small values and compressing large **ONES** so a wide dynamic range fits in one window. When `LOGSCALE` is active, the window also draws power-of-2 markers (1, 2, 4, 8, …) along the amplitude axis.

`LOGSCALE` is **not a decibel mode**. There is no dB scale, no dB markers, and no keyword that produces one — the markers are powers of 2, and the underlying math is log2 of the magnitude.

```
1 DEBUG(`FFT Spectrum SIZE 512 256 SAMPLES 512 LOGSCALE)
2 DEBUG(`Spectrum 'Signal' 3 $7FFF_FFFF 256 0 1 $00FF00)   ' MAG 3
```

Reading the spectrum: bins and frequency

The horizontal axis is **frequency bin number**, left to right, from `FFTfirst` to `FFTlast`. The vertical axis is magnitude. The window itself does **not** label the axis in Hz — it knows nothing about your sample rate. Converting a bin to a frequency is a calculation you do yourself.

If you feed the window samples at a known rate, each bin corresponds to a fixed frequency:

```
frequency of bin k  =  k x (sample_rate / N)
```

So with a 1024-point transform fed at 10 kHz, the bins are spaced `10000 / 1024 ~ 9.77 Hz` apart, and bin 100 sits at about 977 Hz. The highest bin, `N/2 - 1`, sits just below the Nyquist frequency `sample_rate / 2`; signal content above Nyquist aliases down into the displayed range. Choosing the bin range with `SAMPLES N first last` lets you zoom into the band you care about, but the bin-to-Hz arithmetic — and any Hz labeling you want — is yours to add in your own program or notes.

Clearing and saving

Three runtime commands work in the feed stream:

- `` `CLEAR `` — erases the display and resets the sample buffer, so the next spectrum is built from fresh samples rather than blending with what was already collected.
- `` `SAVE 'name' `` — saves the current window image to `name.bmp` on the host.
- `` `CLOSE `` — closes this window and frees its resources.

```
DEBUG(`Spectrum CLEAR)
DEBUG(`Spectrum SAVE 'spectrum')        ' writes spectrum.bmp
```

Packing samples

For low-resolution data you can reduce the serial traffic by packing several samples into each transmitted value. Add one packing keyword to the creation line; the window then unpacks each value it receives into multiple samples. The formats name the container size and the bits per sample:

`LONGS_1BIT`, `LONGS_2BIT`, `LONGS_4BIT`, `LONGS_8BIT`, `LONGS_16BIT`, `WORDS_1BIT`, `WORDS_2BIT`, `WORDS_4BIT`, `WORDS_8BIT`, `BYTES_1BIT`, `BYTES_2BIT`, `BYTES_4BIT`.

For example, `LONGS_8BIT` unpacks each long you send into four 8-bit samples. Packing is most useful for the LOGIC and SCOPE windows where samples are naturally narrow; for full-range FFT input it is rarely needed.

A complete example: a multi-tone spectrum, no hardware

This program needs nothing but a P2 board and the host. It synthesizes a signal in software — three sine tones summed (think of them as a fundamental and two harmonics of a vibrating machine), plus a little noise — and feeds it to the FFT window so you can see the three tones as three spikes and the noise as a low floor beneath them.

The tones come from the CORDIC `qsin` operator. Each tone has its own phase accumulator; adding a fixed increment to a phase each sample sets that tone's frequency, and the size of the increment relative to a full `$1_0000_0000` turn fixes which bin the spike lands in.

```
CON
  _clkfreq = 180_000_000

  N        = 1024                       ' FFT size (power of 2, 4..2048)

```

continues on next page →

↪ *continued from previous page*

```
PUB main() | p1, p2, p3, s

  ' Create a 512x256 FFT window, 1024-point transform, log amplitude.
  DEBUG(`FFT Spectrum SIZE 512 256 SAMPLES 1024 LOGSCALE)

  ' One channel: label, MAG=0, full scale, 256 tall, baseline 0,
  ' baseline grid line, drawn green.
  DEBUG(`Spectrum 'Signal' 0 $7FFF_FFFF 256 0 1 $00FF00)

  p1 := 0                               ' phase accumulators
  p2 := 0
  p3 := 0
  repeat
    repeat N                            ' one full FFT buffer per pass
      s :=  qsin(20000, p1, $1_0000)    ' tone 1
      s += qsin(12000, p2, $1_0000)    ' tone 2
      s += qsin( 6000, p3, $1_0000)    ' tone 3
      s += (GETRND() & $FFF) - $800     ' +/- noise
      DEBUG(`Spectrum `(s))             ' feed one sample

      p1 += $0080_0000                  ' lowest tone
      p2 += $0140_0000                  ' middle tone
      p3 += $0300_0000                  ' highest tone
    waitms(20)
```

`qsin(length, angle, twopi)` returns a CORDIC sine: `length` is the amplitude, `angle` is the current phase, and `twopi` is the value that represents one full turn — here `$1_0000`, so the phase wraps every 65,536 counts. Summing three of them, scaling the noise down with a mask, and feeding the result one sample at a time produces a spectrum with three clear peaks at the three increment-determined bins.

The bin each tone lands in is set by its phase increment: a larger increment advances the phase faster, which is a higher frequency, which is a higher bin. Change an increment and watch the corresponding spike slide along the axis.

> **Extension (real hardware).** To analyze a real signal instead of a synthetic one, read an ADC-configured smart pin in the sample loop and feed its value in place of the `qsin` sum. Everything else — the window, the channel, the redraw — stays the same.

Where you'd use this

In computer science and computer engineering, the FFT window is the tool for **audio and DSP analysis** — measuring harmonics, total harmonic distortion, and noise floors — and for **condition monitoring**, identifying the frequencies a machine or circuit is producing.

On an embedded project, you reach for it to find a motor or bearing's vibration signature, to measure power-line harmonics and THD, to hunt an EMI noise source by its frequency, or to identify a mechanical or electrical resonance. The three synthetic tones above stand in for exactly that kind of content — real components such as a fundamental and two harmonics.

Bandwidth fit: vibration and harmonic work lives at low sample rates, where a buffered block transforms cleanly — an ideal **FIT**. Live full-rate audio is tempered: feed it in buffered or decimated blocks rather than as a continuous stream (see Chapter 7).

Extension (real hardware): read an ADC-configured smart pin in the sample loop and feed its value in place of the `qsin` sum — the window, channel, and redraw stay the same.

Considerations

- **The window is fixed; there is no choice of window function.** Every transform is Hanning-windowed internally. Do not look for a `WINDOW` keyword or alternate window types — there are none.
- **Amplitude is arbitrary units, not dB.** `LOGSCALE` is a log2 compression with power-of-2 markers, and `MAG` is a power-of-2 gain. Neither produces decibels.
- **The frequency axis is yours to compute.** The window plots bins, not Hertz. Bin `k` is at `k x sample_rate / N`; if you want Hz labels, you add them.
- **One FFT per channel per redraw.** Each channel runs its own transform, so spectra for several channels cost proportionally more work per frame; use `RATE` to redraw less often when feeding many channels or large `N`.
- **Collect a full block before the first spectrum appears.** The window waits until it has `N` samples before drawing, so there is a brief fill delay at startup and after `` `CLEAR ``.
- **Pick `N` for the resolution you need.** Larger `N` gives finer bin spacing (better frequency resolution) at the cost of a longer block to fill and more work per transform. The maximum is **2048**.

When to use FFT

- **FFT** — you care about *which frequencies* are present: tones, harmonics, resonances, noise floor.
- **SCOPE** (Chapter 8) — you care about the *waveform over time*: shape, timing, transients.
- **SPECTRO** (Chapter 10) — you care about *how the spectrum changes over time*: a scrolling waterfall built from the same FFT, one column per transform.

FFT and SPECTRO share the same transform and the same SAMPLES/MAG/range configuration; FFT shows the current spectrum as a graph, SPECTRO shows a history of spectra as a color-coded waterfall.

Try it

Start with the multi-tone example above. Then:

1. Change one phase increment and watch its spike move; double an increment and confirm the spike lands roughly twice as far along the axis.
2. Add `SAMPLES 1024 0 200` to zoom into the low end where your tones sit, and see them spread across the full width.
3. Declare a second channel with a different color and baseline, feed it a single pure tone, and compare the clean spike against the noisy three-tone trace.
4. Toggle LOGSCALE off and on to see the noise floor rise into view under log scaling, then raise MAG on the channel to lift weak content further.

You will have used creation config, the SAMPLES size and bin range, a channel declaration with color and grid, and both amplitude controls together — a complete software-only spectrum analyzer in a few dozen lines.

Chapter 10: The SPECTRO Window

Spectrogram Waterfall

The SPECTRO window shows how a signal's frequency content changes over time. It runs an FFT on a sliding window of samples, turns each transform into one line of colored pixels — one pixel per frequency bin, color set by that bin's magnitude — and scrolls the line stack so the newest spectrum appears at the edge and older spectra drift away. The result is a *waterfall*: frequency along one axis, time along the other, intensity carried by color.

This is the difference between SPECTRO and the FFT window of Chapter 9. The FFT window draws one spectrum at a time as a magnitude-versus-frequency graph and redraws it on every update; you see *now*, and nothing else. SPECTRO keeps a history. A tone that slides up in pitch traces a diagonal streak; a steady harmonic draws a straight line; a transient flashes as a short band. SPECTRO is **single channel** — it analyzes one stream of samples, unlike the FFT window's up-to-eight overlaid channels.

You create one SPECTRO window per `DEBUG(`SPECTRO ...)` declaration, name it, and feed it samples by that name. This chapter covers creating the window, feeding it, choosing scroll direction and update rate, mapping magnitude to color, and the runtime commands.

> Keyboard and mouse input (`PC_KEY`, `PC_MOUSE`) work in the SPECTRO window, but they share one mechanism across every window type and are covered together in Chapter 12. This chapter is about the display.

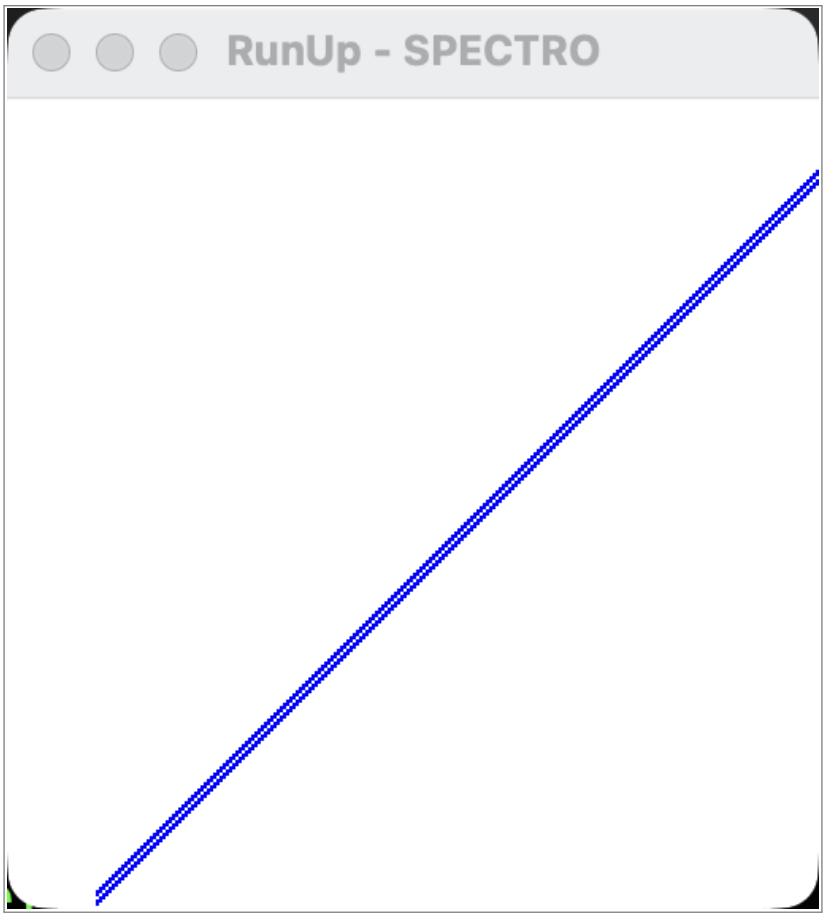

Figure 10.1. The SPECTRO window as a motor run-up: rising vibration frequency draws a diagonal streak down the waterfall.

Creating a SPECTRO window

You create and configure the window in a single `DEBUG` statement. The first token after the backtick is the window type (`SPECTRO`); the second is a name you choose. You feed it afterward by that name:

```
DEBUG(`SPECTRO Wfall SAMPLES 512 DEPTH 256 RANGE $40000 LUMA8X)
DEBUG(`Wfall `(sample))      ' feed it one sample by name
```

The configuration keywords for the creation line:

Keyword	Arguments	Default	What it sets
TITLE	'text'	SPECTRO	The window's title-bar text
POS	left top	auto	Screen position of the window, in pixels
SAMPLES	count	512	FFT size; a power of two, **4–2048**
DEPTH	pixels	256	Time-history depth (the scrolling dimension), **1–2048**
RANGE	value	$7FFFFFFF	Magnitude ceiling — the bin magnitude that maps to full color, **1–$7FFFFFFF**
RATE	samples	SAMPLES/8	Samples taken in between display updates, **1–2048**
TRACE	mode	15	Scroll direction and scroll-enable (see "Scroll direction")
MAG	shift	0	Magnitude pre-scale; multiplies FFT output by 2^{}shift, **0–11**
DOTSIZE	x [y]	1	Pixel scaling; one value sets both axes, two set them separately, **1–16**
color mode	—	LUMA8X	One color-mode keyword (see "Color mapping")
LOGSCALE	—	linear	Logarithmic magnitude scaling instead of linear
HIDEXY	—	off	Hides the coordinate readout

SAMPLES is the first decision. It sets the FFT size, and the FFT produces SAMPLES/2 frequency bins spanning DC up to the Nyquist frequency (half your effective sample rate). A SAMPLES 512 window analyzes 512 points into 256 bins; SAMPLES 2048 gives 1024 bins. SAMPLES is rounded to a power of two — values that are not powers of two are reduced to the next lower power, and the range is clamped to 4–2048.

SAMPLES and DEPTH set the two axes. The frequency axis is SAMPLES/2 bins long. The time axis is DEPTH pixels long — that is how many past spectra stay on screen before scrolling off. Which axis is horizontal and which is vertical depends on TRACE, covered below.

> SAMPLES **is the FFT size, not a keyword named** FFT_SIZE**.** There is no FFT_SIZE, no OVERLAP, and no windowing option — the analysis window is a fixed Hanning window, identical to the FFT window's. You control the transform through SAMPLES, RANGE, MAG, and LOGSCALE.

Feeding samples

After the window exists, send signed sample values by its name. Each value is one time-domain sample. Send the *value* with `` `() `` — not a display formatter, which would print visible digits rather than deliver a number:

```
DEBUG(`Wfall `(sample))
```

Internally the window stores samples in a circular buffer. It does nothing visible until it has collected a full SAMPLES window; from then on it runs an FFT over the most recent SAMPLES samples whenever the rate counter says it is time, and plots the result as one line. You keep feeding samples; the window decides when to transform and draw.

You can send several samples in one DEBUG **CALL** by listing them, and you can pack multiple samples per long for higher throughput. SPECTRO uses the same 12-mode packing scheme as the other sampling windows: a packing keyword on the feed selects how many samples each long carries and whether they are sign-extended.

```
DEBUG(`SPECTRO Pk SAMPLES 512 RANGE $4000 LONGS_8BIT LUMA8X)
' four signed bytes -> four samples
DEBUG(`Pk `($7F | $40 << 8 | $C0 << 16 | $10 << 24))
```

The packing keywords are LONGS_1BIT, LONGS_2BIT, LONGS_4BIT, LONGS_8BIT, LONGS_16BIT (sign-extended), and WORDS_1BIT/2BIT/4BIT/8BIT and BYTES_1BIT/2BIT/4BIT (zero-extended). LONGS_8BIT carries four 8-bit signed samples per long, a 4× bandwidth gain over sending one sample per long.

Scroll direction — TRACE

TRACE does two jobs in one value, **0–15**:

- **Bits 0–2** select one of eight trace directions.
- **Bit 3** enables scrolling. With bit 3 clear (TRACE 0–7) the window *wraps* — new lines overwrite from the opposite edge with no scrolling. With bit 3 set (TRACE 8–15) the bitmap *scrolls* one line per update, the classic waterfall.

The direction bits also decide which axis is frequency and which is time. Directions 0–3 lay frequency along the horizontal axis and scroll the time history vertically; directions 4–7 put time on the horizontal axis and frequency vertical. The window swaps its width and height accordingly when it sizes itself.

The default is TRACE 15 — direction 7 with scrolling on. For a downward-scrolling waterfall with frequency across the top, use TRACE 8:

```
DEBUG(`SPECTRO Wfall SAMPLES 512 DEPTH 256 TRACE 8 RANGE $40000 LUMA8X)
```

For a vertical waterfall scrolling sideways — frequency up the side, time advancing horizontally — use a direction in the 4–7 group with bit 3 set, for example `TRACE 12`:

```
DEBUG(`SPECTRO Vert SAMPLES 256 DEPTH 400 TRACE 12 ...
        RANGE $20000 HSV16X LOGSCALE)
```

Set scrolling on (`TRACE` 8–15) for a waterfall. Values 0–7 wrap in place, which overwrites old history rather than scrolling it away. The scrolling forms are what give SPECTRO its waterfall behavior.

Update rate — RATE

`RATE` is the number of samples the window collects between display updates. It does not change the FFT size; it controls how often a new line is drawn, and therefore how fast the waterfall scrolls. Smaller `RATE` means more updates per second and faster scrolling at higher CPU cost; larger `RATE` means slower scrolling.

The default is `SAMPLES`/8 — for a 512-point FFT, an update every 64 samples. The effective scroll rate in lines per second is your sample feed rate divided by `RATE`. Set `RATE` to control how much real time each line of the display represents.

```
DEBUG(`SPECTRO Slow SAMPLES 2048 DEPTH 200 RATE 512 TRACE 8 ...
        RANGE $80000 LUMA8X)
```

`RATE` accepts **1–2048**.

Color mapping

SPECTRO turns each bin's magnitude into a color. First the magnitude is scaled: `RANGE` is the magnitude that maps to full intensity, so it sets the display's sensitivity. A single value — `RANGE $40000` — is the ceiling; magnitudes at or above it saturate, magnitudes below scale proportionally to 0–255. There is no floor, no second value. Lower `RANGE` to make weak signals brighter; raise it when strong signals wash out.

```
DEBUG(`SPECTRO Sens SAMPLES 512 RANGE $8000 MAG 4 LOGSCALE LUMA8X)
```

Two more controls shape the magnitude before color:

- `MAG shift` multiplies the FFT output by 2^{}shift (a 0–11 bit pre-shift), raising low-level signals before scaling.
- `LOGSCALE` applies logarithmic magnitude scaling instead of linear, which compresses a wide dynamic range so faint detail stays visible alongside strong peaks.

The scaled 0–255 value then drives a color-mode keyword. The modes SPECTRO accepts on its creation line are the luminance and 16-bit HSV families:

Keyword	Encoding	What it does
`LUMA8`	8-bit luminance	Black-to-color ramp; magnitude sets brightness
`LUMA8W`	8-bit luminance	White-to-color ramp (inverted)
`LUMA8X`	8-bit luminance	Extended-range luminance — the default
`HSV16`	16-bit HSV	Hue/saturation/value; magnitude in value
`HSV16W`	16-bit HSV	White variant
`HSV16X`	16-bit HSV	Extended range

`LUMA8X` is the default if you name no mode. The luminance modes render a brightness ramp — the natural "heat map" look for a single magnitude. The HSV16 modes encode phase as well: the FFT's per-bin phase angle is folded into the hue while magnitude drives the value, so an HSV16 waterfall shows both how strong each frequency is and its phase relationship. Choose a luminance mode when magnitude is all you need; choose an HSV16 mode when phase matters.

> The theory-of-operations lists additional color-encoding constants (HSV8, RGBI8, RGB8/16/24, and LUT modes) shared across the display infrastructure, but SPECTRO's own configuration parser accepts only the LUMA8 and HSV16 families above. Those are the modes to use here.

Runtime commands — CLEAR, SAVE, and CLOSE

Three keyword commands work at runtime, sent by the window's name:

- `` `CLEAR `` — clears the display, resets the sample buffer (so the window waits for a fresh full window before drawing again), and resets the trace position to its starting edge.
- `` `SAVE `` — saves the current window image to a file on the host.
- `` `CLOSE `` — closes this window and frees its resources.

```
DEBUG(`Wfall CLEAR)
DEBUG(`Wfall SAVE)
```

Use `` `CLEAR `` to start a new capture cleanly — after it, the next `SAMPLES` samples refill the buffer before anything new is drawn.

A complete software-only example: a motor run-up

This program needs no wiring. It synthesizes a signal that stands in for the **vibration of a motor as it runs up to speed**. A spinning motor vibrates most strongly at its shaft-rotation frequency; as it accelerates from rest to full speed, that tone climbs. We make exactly that with the CORDIC: a sine tone whose frequency rises block by block. Fed to a downward-scrolling SPECTRO, the rising vibration draws a **diagonal streak** down the waterfall — the run-up captured as a picture.

```
CON
  _clkfreq = 200_000_000

PUB main() | i, phase, ainc, sample
  ' One scrolling spectrogram, 512-point FFT, 256 lines of history.
  DEBUG(`SPECTRO RunUp SAMPLES 512 DEPTH 256 RANGE $40000 ...
         RATE 512 TRACE 8 LUMA8X)

  phase := 0
  ainc  := 30_000              ' shaft frequency at rest (a low tone)

  repeat
    ' Feed one 512-sample FFT window at the current speed, then accelerate.
    repeat i from 1 to 512
      sample := sine(2000, phase)
      phase += ainc            ' advance the synthesized vibration tone
      DEBUG(`RunUp `(sample))
    ainc += 20_000  ' the motor speeds up -> higher tone -> diagonal streak
    if ainc > 1_000_000
      DEBUG(`RunUp CLEAR)      ' reached top speed: clear and run up again
      ainc := 30_000

PRI sine(amp, angle) : y
  ' amp * sin(angle), via the CORDIC.
  ' angle $0000_0000..$FFFF_FFFF spans one full circle.
  ORG
    QROTATE amp, angle         ' X = amp, Y = 0 (no SETQ)
    GETQY   y                  ' Y result = amp * sin(angle)
  end
```

`sine()` uses **QROTATE** to rotate the point (amp, 0) by `angle`; **GETQY** returns the Y component, which is `amp x sin(angle)`. Stepping `phase` by `ainc` each sample produces a tone whose frequency is set by `ainc` — the stand-in for shaft speed; raising `ainc` after every 512-sample block accelerates the

motor, and the waterfall records the climb as a diagonal. The Hanning window is applied inside the FFT automatically; you supply only the samples.

Hold `ainc` constant and the motor runs at a steady speed — the same frequency every block draws a straight vertical streak (with `TRACE 8`). A structural resonance the machine passes through on its way up shows as a bright spot where the climbing tone crosses that fixed frequency — exactly the resonance-crossing a machine-health engineer watches for during run-up and coast-down.

Where you'd use this

In computer science and computer engineering, SPECTRO is the tool for **spectral monitoring over time** — watching how a signal's frequency content evolves — in narrowband RF and communications and in acoustics and speech analysis.

On an embedded project, its natural home is **machine-health monitoring**: trending a motor or bearing's vibration spectrum so you can see a fault band grow, watching resonance crossings during run-up and coast-down (as here), or following a narrowband or voice signal over time.

Bandwidth fit: vibration and acoustic monitoring live at **SUB**-10 kHz and play out over seconds to minutes — low sample rate, long duration — which is exactly what the link and the waterfall want. A full-rate RF or music spectrum does not **FIT** and is out.

Extension (real hardware): replace the synthetic `sine()` with real samples from an accelerometer or microphone — read an ADC or I²S input in the feed loop — and the waterfall shows live machine vibration.

Considerations

- **Single channel.** SPECTRO analyzes one sample stream. To compare several signals in the frequency domain at one instant, the FFT window (Chapter 9) overlays up to eight channels; SPECTRO trades that for time history on one channel.
- **The window is a fixed Hanning window.** You cannot select a different window function or set overlap. Spectral leakage is what Hanning gives you — the same as the FFT window.
- `SAMPLES` **trades frequency resolution against time resolution.** Large `SAMPLES` (e.g. 2048) gives fine frequency detail but each line averages more time; small `SAMPLES` (e.g. 128) reacts fast in time but resolves frequency coarsely. Pick to match whether you care more about *which* frequency or *when*.
- `RATE` **sets scroll speed,** `DEPTH` **sets how much history is visible.** They are independent: `RATE` is samples-per-line, `DEPTH` is lines-on-screen. Together they determine the real-time span shown.
- **Tune visibility with** `RANGE`**, then** `MAG`/`LOGSCALE`**.** Start by setting `RANGE` near your expected peak magnitude. If weak detail is still too dark, add `MAG` to pre-amplify or `LOGSCALE` to compress the dynamic range.
- **Pack for throughput.** A high sample feed rate over the DEBUG link benefits from `LONGS_8BIT` or similar; packing multiplies how many samples each long carries.

Try it

Start from the run-up example. Then:

1. **Switch axes.** Change `TRACE 8` to `TRACE 12` and watch the waterfall scroll sideways with frequency up the side.
2. **Add a second source.** Sum a second `sine()` at a fixed frequency into each sample so a steady horizontal line — a second machine running at constant speed — sits alongside the moving diagonal; toggle it on and off per block to see it appear and vanish:

```
sample := sine(1500, p1)
p1 += 200_000
if t & 1
  sample += sine(1500, p2)   ' second tone toggles per block
  p2 += 600_000
```

3. **Reveal weak detail.** Lower `RANGE`, then add `LOGSCALE`, and compare how much more of the spectrum becomes visible.

You will have used `SAMPLES`, `RANGE`, `TRACE`, `RATE`, a color mode, and `CLEAR` together — and built a working spectrogram with no hardware beyond the P2 board.

Chapter 11: The MIDI Window

Piano-Keyboard Display

The MIDI window draws an on-screen piano keyboard and lights its keys in response to MIDI note messages. You feed it raw MIDI bytes — the same Note-On and Note-Off bytes a synthesizer or controller would send — and it illuminates the matching key, filling it from the bottom up in proportion to the note's velocity. It is the window you reach for when you are debugging a MIDI implementation, watching a sequencer or synthesizer engine you wrote, or visualizing a performance: instead of reading a stream of hex bytes, you see the notes appear on a keyboard.

The window is purely a *display*. It does not produce sound and it does not read a MIDI port for you. You generate the MIDI bytes in software and send them with `DEBUG()`; the window parses them and updates the keyboard. Every example in this chapter runs on a bare P2 with no external MIDI hardware.

> Keyboard and mouse input (`PC_KEY`, `PC_MOUSE`) work in the MIDI window as in every other window. They share one mechanism across all window types, so they are covered together in Chapter 12. This chapter is about the keyboard display.

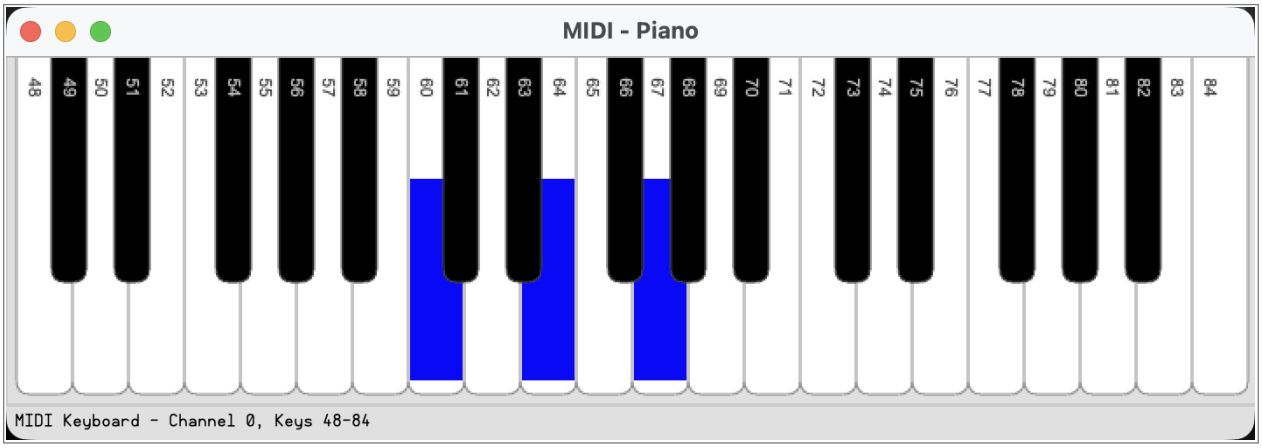

Figure 11.1. The MIDI window showing notes lit on a piano keyboard.

Creating a MIDI window

You create and configure the window in a single `DEBUG` statement. The first token after the backtick is the window type (`MIDI`); the second is a name you choose. You feed the window afterward by that name:

```
DEBUG(`MIDI Piano SIZE 6 RANGE 48 84 CHANNEL 0)   ' create, named "Piano"
DEBUG(`Piano $90 60 96)  ' feed it by name: note-on
```

The configuration keywords you can add to the creation line:

Keyword	Arguments	Default	What it sets
`TITLE`	`'text'`	`MIDI`	The window's title-bar text
`POS`	`left top`	auto	Screen position of the window, in pixels
`SIZE`	`multiplier`	`4`	Key-size multiplier, **1–50**
`RANGE`	`first last`	`21 108`	First and last MIDI note to display, each **0–127**
`CHANNEL`	`channel`	`0`	The single MIDI channel to display, **0–15**
`COLOR`	`white_active` `black_active`	cyan, magenta	Lit-key colors (white key, then black key), each `$RRGGBB`

A few things to know about these:

- `SIZE` is a multiplier, not a pixel count. The white-key width is `8 + SIZE x 4` pixels, so `SIZE 4` gives 24-pixel keys and `SIZE 50` gives 208-pixel keys. The whole window sizes itself from this and from the note range; you do not set a width and height directly.
- `RANGE` selects which slice of the 128 MIDI notes is drawn. The default `21 108` is the standard 88-key piano (A0 to C8). You can show any subset — `RANGE 60 72` is one octave from middle C — or the full `RANGE 0 127`.
- `CHANNEL` is a single channel, 0–15. The window displays notes only on that channel and ignores all others; it does not have an "all channels" mode. MIDI channels are numbered 1–16 on instruments, so channel 1 is `0` here and channel 16 is `15`.
- `COLOR` takes exactly two color values: the first is the lit color for white keys, the second for black keys. Unlit white keys are always white and unlit black keys are always black; only the *active* fill color is configurable.

```
' One octave, large keys, green/orange lit colors, channel 0
DEBUG(`MIDI Keys SIZE 8 RANGE 60 72 CHANNEL 0 COLOR $00FF00 $FF7F00)
```

Each key is labelled with its MIDI **note number** (0–127), drawn rotated along the key. The labels are note numbers, not musical note names — note 60 reads `60`, not "C4".

The MIDI bytes the window understands

Everything you send after the creation line is a stream of bytes — plain numeric values, sent the same way you send command codes to other windows. The window runs a MIDI parser over that byte stream. It recognizes exactly two messages:

Message	Status byte	Then	Then
Note-On	`$9n` (n = channel)	note number, 0–127	velocity, 0–127
Note-Off	`$8n` (n = channel)	note number, 0–127	velocity, 0–127

`n` is the channel nibble. A Note-On on channel 0 is `$90`; on channel 5 it is `$95`. The window acts on a message only when its channel nibble matches the configured `CHANNEL`.

How the parser reads the stream:

- A byte with its top bit set (`$80`–`$FF`) is a **status byte**. It resets the parser and selects what comes next. `$9n` starts a Note-On; `$8n` starts a Note-Off (both only if `n` matches `CHANNEL`).
- A byte with its top bit clear (`$00`–`$7F`) is a **data byte** — a note number, then a velocity.
- A Note-On sets the note's velocity to the value you send. A Note-Off sets it to the negative of the value, which the renderer treats as "off." Either way the keyboard redraws after the velocity byte arrives.

Running status. After a status byte, the parser stays in that message type and keeps reading note/velocity pairs until a new status byte arrives. So one `$90` followed by several note/velocity pairs plays several notes on — you do not repeat the `$90`:

```
1 DEBUG(`Piano $90 60 80 64 80 67 80)   ' three note-ons: C, E, G
```

That is three Note-On messages expressed as one status byte plus three pairs.

Velocity and the lit fill

Velocity (0–127) controls how far up the key the lit color fills. A key lit at velocity 127 fills completely; at velocity 64 it fills about halfway; at low velocity only a sliver at the bottom lights. The fill always grows from the bottom of the key upward, so a row of held notes reads like a bar chart of how hard each was struck. White keys fill with the first `COLOR` value (default cyan); black keys with the second (default magenta).

Sending notes

To light a key, send a Note-On for it; to clear it, send a Note-Off for the same note:

```
DEBUG(`Piano $90 60 96)   ' middle C (note 60) on, velocity 96
waitms(500)
DEBUG(`Piano $80 60 0)    ' middle C off
```

When a note number comes from a variable, send its value with `` `() `` so it goes out as a data byte rather than visible digits:

```
DEBUG(`Piano $90 `(note) `(vel))   ' note-on with variable note and velocity
```

This is the same distinction as in the other windows: `` `(note) `` sends the *value* of `note` as a byte the parser consumes, whereas a formatter such as `` `udec_(note) `` would render the digits as text — which the MIDI window does not accept and would ignore as a string.

Clearing and saving

Three runtime keyword commands round out the set:

- `` `CLEAR `` — resets every key to off (clears all stored velocities) and redraws an empty keyboard. Use it between takes, or to recover if a Note-Off was missed and a key is stuck lit.
- `` `SAVE `` — saves the current window image to a file on the host.
- `` `CLOSE `` — closes this window and frees its resources.

```
DEBUG(`Piano CLEAR)   ' all keys dark again
```

A complete software-only example

This program needs nothing but a P2 and the host running `pnut_term_ts`. It generates its own MIDI bytes: it plays a C-major scale one note at a time, then a C-major chord using running status, then clears the keyboard.

```
CON
  _clkfreq = 200_000_000

PUB main() | i, note
  DEBUG(`MIDI Piano SIZE 6 RANGE 48 84 CHANNEL 0)

  ' --- C major scale, one note at a time, on channel 0 ---
  ' note-on  = $90, note, velocity ;  note-off = $80, note, velocity
  repeat i from 0 to 6
```

continues on next page →

↪ continued from previous page

```
    note := WORD[@scale][i]
    DEBUG(`Piano $90 `(note) 96)       ' note-on, velocity 96 -> key fills
    waitms(300)
    DEBUG(`Piano $80 `(note) 0)        ' note-off -> key clears
    waitms(60)

  ' --- C major chord via running status: one $90, then note/vel pairs ---
  ' C E G, all velocity 80, held together
  DEBUG(`Piano $90 60 80 64 80 67 80)
  waitms(1000)
  DEBUG(`Piano $80 60 0 64 0 67 0)     ' release all three (running status)
  waitms(300)

  DEBUG(`Piano CLEAR)                  ' reset every key

  repeat                              ' keep the window open

DAT
scale WORD 60, 62, 64, 65, 67, 69, 71  ' C D E F G A B (MIDI note numbers)
```

Watch the keyboard: each scale note lights one key for 300 ms, then the three chord keys light together for a second, then the keyboard goes dark.

To see velocity at work, hold one note and raise the velocity each pass — the lit fill climbs higher each time:

```
CON
  _clkfreq = 200_000_000

PUB main() | vel
  DEBUG(`MIDI Keys SIZE 4 RANGE 21 108 CHANNEL 0 COLOR $00FF00 $FF7F00)

  vel := 24
  repeat 5
    DEBUG(`Keys $90 60 `(vel))         ' middle C on at the current velocity
    waitms(400)
    DEBUG(`Keys $80 60 0)              ' off
    waitms(150)
    vel += 25
    if vel > 127
      vel := 127
```

continues on next page →

↪ *continued from previous page*

```
16
17   repeat                                  ' keep the window open
```

Where you'd use this

The honest answer is narrow: the MIDI window is for **music technology and MIDI protocol work**. Its job is to show Note-On / Note-Off activity on a keyboard, and that is the whole of it.

On an embedded project, that means debugging a synth or sequencer engine you are writing, verifying the note output of a MIDI controller, or visualizing a generative-music algorithm as it plays.

Bandwidth fit: MIDI is a slow, event-driven stream — comfortably inside the link with room to spare.

Extension (real hardware): feed real MIDI bytes from a UART smart pin into the window in place of the hardcoded notes, and it shows a live instrument's playing.

If you are not building MIDI software, this is not your window. Status values belong in TERM (Chapter 3), a changing value in PLOT (Chapter 5), and digital event timing in LOGIC (Chapter 6).

Considerations

- **Only Note-On and Note-Off are recognized.** The parser acts on `$9n` and `$8n` and nothing else. Program Change, Control Change, Pitch Bend, Channel and Polyphonic Aftertouch, System Exclusive, and System Real-Time messages are not supported. If your software emits them, they will not move any key — and because their status bytes have the top bit set, each one simply resets the parser to wait for the next Note-On or Note-Off.
- **A velocity-0 Note-On is not a Note-Off.** Many MIDI sources end a note with a Note-On at velocity 0 instead of a real Note-Off. This window does not treat that as a release: the key would store velocity 0 and read as off, but to turn a lit key off reliably you must send a proper Note-Off (`$8n`) for it. When you generate the bytes yourself, always pair each `$90` note-on with an `$80` note-off.
- **One channel at a time.** The window shows exactly the channel set by `CHANNEL`; notes on other channels are ignored. To watch several channels at once, open one MIDI window per channel, each with its own `CHANNEL` value.
- **Choose the range to the job.** A narrow `RANGE` makes each key larger and the window narrower for the notes you care about; the full `RANGE 21 108` (88 keys) or `RANGE 0 127` is wide. Combine `RANGE` with `SIZE` to fit your display.
- **Note numbers, not note names.** Keys are labelled with MIDI note numbers (0–127). Middle C is 60; concert-pitch A is 69. There is no note-name labelling and no key-naming option.

Try it

Start from the scale-and-chord example. Then build a short melody as a `DAT` table of note/duration pairs and step through it, sending a Note-On, waiting the duration, and sending the Note-Off. Add a second voice by sending a sustained bass note on the same channel while the melody plays over it — you will see the held key stay lit (filled to its velocity) while the melody keys come and go above it. Finally, set `COLOR` to a pair of your own colors and lower the velocity on the bass note to confirm the lit fill tracks velocity from the bottom up.

> **See also.** Keyboard and mouse input in any window, including this one, is covered in Chapter 12.

Part III

Integration

The window chapters each drove one window in one direction: your program produces output, the window shows it. Real debugging asks for more, and Part III covers the four things you reach for once the individual windows are familiar.

Chapter 12 reverses the flow — `PC_KEY` and `PC_MOUSE` read the host's keyboard and mouse back through the same **DEBUG** link, turning any window into a control surface. Chapter 13 makes that link carry more: packed-data modes fold many small samples into a single `DEBUG()` **CALL** so a high-rate capture keeps up. Chapter 14 runs several windows at once and reaches into PASM2, driving the same windows from assembly in their own cog. Chapter 15 combines the pieces into **control and status panels** — instrument readouts you watch and on-screen surfaces you operate — built from the windows you already know, in a few dozen lines of code.

None of these is a new window. Each is a technique that combines the windows you already know.

Chapter 12: Bidirectional Control

Keyboard and Mouse

Every chapter so far has sent data one way: your P2 produces output, a DEBUG display window shows it. This chapter reverses the direction. Using two commands — `PC_KEY` and `PC_MOUSE` — your running program reads the host PC's keyboard and mouse back through the same **DEBUG** link, so the window you opened for output becomes a control surface as well.

The mechanism is shared. `PC_KEY` and `PC_MOUSE` work on any display window — TERM, PLOT, SCOPE, BITMAP, all of them — because you address the input to a window by name and the host reports the state of that window. The window must have focus for input to be noticed: keypresses and wheel events go to whichever window the host user has clicked into.

Both commands follow the same shape: you pass a **pointer to a buffer in hub RAM**, and the host writes the current input state into that buffer. They do not return a value. You read the result afterward from the variable you pointed at:

```
PUB main() | key, mouse[7]
  DEBUG(`TERM Console SIZE 40 10)

  DEBUG(`Console PC_KEY(@key))            ' host writes key code into key

  DEBUG(`Console PC_MOUSE(@mouse))        ' host fills 7 longs

  repeat
```

One rule governs both: **`PC_KEY` and `PC_MOUSE` must be the last command in their `DEBUG()` statement.** You can send output earlier in the same statement, but the input command comes last. In Spin2 the buffer lives in hub RAM, so you pass its address with `@` — `@key`, `@mouse`. (In PASM the buffer is a cog register and you pass it with `#`; this chapter is Spin2.)

PC_KEY — reading the keyboard

`PC_KEY(pointer_to_long)` takes the address of a single long. The host writes the most recent keypress that occurred within the last 100 ms into that long, and writes **0 when no key was pressed**. It is not a function that returns the key — this is wrong:

```
' WRONG - PC_KEY does not return a value
key := DEBUG(`Console PC_KEY)
```

The correct form passes a pointer and then reads the long:

```
DEBUG(`Console PC_KEY(@key))   ' host fills key
case key                        ' now read it
  ...
```

The window must have focus for the keypress to be seen. Because the host reports the keypress from the last 100 ms, poll at least that often if you do not want to miss keys.

Key codes

The long holds one of these values. Printable characters arrive as their ASCII code; a small set of navigation and editing keys arrive as low codes:

Code	Key	Code	Key
0	no keypress	8	Backspace
1	Left Arrow	9	Tab
2	Right Arrow	10	Insert
3	Up Arrow	11	Page Up
4	Down Arrow	12	Page Down
5	Home	13	Enter
6	End	27	Esc
7	Delete	32–126	Space through ~ (all symbols, digits, letters)

That is the complete list the source defines. The four arrows are 1–4 — not $C2/$C3 or any other encoding. There are no function-key codes and no modifier state: PC_KEY reports a single key per poll, with no separate Shift / Ctrl / Alt flags. A capital A arrives as code 65 and a lowercase a as 97, but you cannot read "Ctrl is held" independently of a key.

Example: arrow keys adjust a value

This program opens a TERM window and lets the arrow keys nudge a number up and down. Up/Down change it by one; Left/Right by ten. The value is redrawn only when a key is actually pressed:

```
CON _clkfreq = 200_000_000

PUB main() | key, value
  DEBUG(`TERM Adjust SIZE 32 6 TITLE 'Arrow Keys Adjust')
  value := 50
  show(value)
  repeat
```

continues on next page →

↪ continued from previous page

```
    key := 0
    DEBUG(`Adjust PC_KEY(@key))
    case key
      3: value += 1                         ' Up arrow
      4: value -= 1                         ' Down arrow
      1: value -= 10                        ' Left arrow
      2: value += 10                        ' Right arrow
    if key
      show(value)
    waitms(20)

PRI show(v)
  DEBUG(`Adjust 0 "Value: " `sdec_(v) 13 "Up/Down +/-1, Left/Right +/-10")
```

Click the window to give it focus, then press the arrow keys. Each `case` arm acts on one key code; the `if key` guard skips the redraw on the polls where no key arrived (`key` is 0). The `key := 0` at the top of each pass clears the variable so a stale value is never re-acted on.

PC_MOUSE — reading the mouse

`PC_MOUSE(pointer_to_7_longs)` takes the address of a **seven-long array**. The host fills all seven with the current mouse state relative to the named window. Declare the buffer as `LONG mouse[7]` and pass `@mouse`:

```
DEBUG(`Watch PC_MOUSE(@mouse))
```

The seven longs, in order, are:

Index	Field	Meaning
`mouse[0]`	xpos	X position within the window. **Negative if the mouse is outside the window** (both xpos and ypos go negative together).
`mouse[1]`	ypos	Y position within the window.
`mouse[2]`	wheel delta	Scroll-wheel change: `0`, or `+1` / `-1` when the wheel moved. The window must have focus.
`mouse[3]`	left button	`0` released, `-1` pressed.
`mouse[4]`	middle button	`0` released, `-1` pressed.
`mouse[5]`	right button	`0` released, `-1` pressed.
`mouse[6]`	pixel	Color at the mouse position, `$00_RR_GG_BB`, or `-1` if the mouse is outside the window.

The position units depend on the window type — for a TERM window they are character column and row; for pixel-based windows like BITMAP they are pixels. (See the per- window hover-coordinate behavior in each window's chapter.)

Buttons are a full-long state, not a bitmask. Each button long is either `0` (released) or `-1` (pressed). Test it directly — `if mouse[3]` is true when the left button is down. Do not mask it:

```
' WRONG - buttons are 0 / -1, not packed bits
if mouse[3] & 1
```

```
' CORRECT
if mouse[3]
```

Because `-1` is all bits set, a bit **TEST** happens to work for the low bit, but it misrepresents the data and is not how the host reports it. Treat each long as a boolean state of one button.

Detect "outside the window" with the position: `mouse[0]` (and `mouse[1]`) go negative when the pointer leaves the window, and `mouse[6]` is `-1` in that case too.

Example: read mouse position and buttons

This program continuously displays the mouse state in a TERM window, clearing and redrawing each pass:

```
CON _clkfreq = 200_000_000

PUB main() | mouse[7]
  DEBUG(`TERM Pointer SIZE 40 8 TITLE 'Mouse State')
  repeat
    DEBUG(`Pointer PC_MOUSE(@mouse))
    DEBUG(`Pointer 0)                                  ' clear + home
    if mouse[0] < 0
      DEBUG(`Pointer "Pointer outside window" 13)
    else
      DEBUG(`Pointer "X: " `sdec_(mouse[0]) "  Y: " `sdec_(mouse[1]) 13)
      DEBUG(`Pointer "Wheel: " `sdec_(mouse[2]) 13)
      DEBUG(`Pointer "Buttons  L:" `sdec_(mouse[3]))
      DEBUG(`Pointer "  M:" `sdec_(mouse[4]) "  R:" `sdec_(mouse[5]) 13)
      DEBUG(`Pointer "Pixel: " `uhex_long_(mouse[6]))
    if mouse[3]                                         ' left button down?
      DEBUG(`Pointer 13 "LEFT DOWN")
    waitms(30)
```

Move the mouse over the window and the position updates; move outside and the program reports it from the negative `xpos`. Press the left button and the buttons read `-1` (shown as `-1` by `` `sdec_ ``), and the `LEFT DOWN` line appears.

Where you'd use this

In computer science and computer engineering, host input turns a DEBUG display window into a **human-in-the-loop control surface** — interactive parameter adjustment and manual **TEST** rigs you drive by hand while the program runs.

On an embedded project, you reach for it to tune PID gains live, to nudge a setpoint, to jog an actuator by hand, or to trigger and label a calibration capture — all without recompiling, using the host keyboard and mouse as a temporary control panel.

Bandwidth fit: input is polled a few tens of times a second; it is negligible against the link budget.

Extension (real hardware): the same `PC_KEY` / `PC_MOUSE` polling that reads the host here can hand its values to real outputs — drive a smart-pin PWM from a tuned gain, step a motor from an arrow key — turning the panel into live control.

Considerations

- **The model is polling, not interrupts.** Nothing is pushed to your program; you ask for the current state each time you issue `PC_KEY` / `PC_MOUSE`. Build the command into your loop and poll at a steady rate.
- `PC_KEY` **reports keys from the last 100 ms.** Poll at least every 100 ms or you can miss a keypress. The examples here poll far faster than that.
- **One key per poll, no modifier state.** `PC_KEY` returns a single code; there is no separate Shift / Ctrl / Alt status and no way to read two keys held together. Design around single-key input.
- **Buttons and "outside" are sentinel values, not flags.** Buttons are `0` / `-1`; position goes negative and `pixel` becomes `-1` when the pointer leaves the window. **TEST** these as whole-long states.
- **Must be the last command in the** `DEBUG()` **statement.** Put any output earlier; end the statement with the input command.
- **Focus matters.** The host user must click the window to give it focus before keypresses and wheel deltas register. The window name in the command selects *which* window's input you read.

Try it

Start from the mouse example and turn the window into a click target: when `mouse[3]` goes from `0` to `-1` (a press edge — track the previous value in a variable so you fire once per click, not every poll), increment a counter and show it. Then combine both commands in one loop: poll `PC_KEY` to

reset the counter on Esc (code 27) and `PC_MOUSE` to count clicks. You will have a single window that reads both input devices, using nothing but the debug link and a bare P2 board.

Chapter 13: Packed Data

Compact High-Rate Transfers

Every element you send to a window travels over the `DEBUG()` serial link. That link is finite. The P2 transmits debug output on pin P62 in 8-N-1 format at the rate set by the `DEBUG_BAUD` symbol, which defaults to `DOWNLOAD_BAUD` — **2 Mbaud**. `pnut_term_ts` is certified at 2 Mbaud, so keep the link there: set `DEBUG_BAUD` explicitly only if you have changed `DOWNLOAD_BAUD` or your clock requires it, and do not drop the DEBUG link to a slow rate such as 115200 — debugging needs the bandwidth. (If you drive the windows from the Spin Tools IDE, confirm it runs at 2 Mbaud.) When your data is *small* — single bits from a logic capture, 8-bit samples from a scope trace — sending one value per element wastes the link: a 1-bit sample carried as a full long spends 32 bits of wire to convey one bit of information.

Packed-data modes fix that. You pack many small values into a byte, word, or long on the P2 side, send the container as a single element, and the host **unpacks** it back into the individual values. A single `DEBUG` element can then carry as many as 32 samples instead of one. The instrument windows — LOGIC, SCOPE, SCOPE_XY, FFT, SPECTRO, and BITMAP — all read the same packed formats, so the same technique raises the effective sample rate of every one of them.

This chapter documents the complete set of packed-data modes, how to feed them, and how to choose one for your data.

Why packing helps

The `DEBUG()` link carries every element you send one after another over the serial connection, so the rate at which you can feed a window is bounded by that link. A high-rate capture — a logic-analyzer trace, a fast scope sweep — generates samples faster than one-element-per-sample transmission can keep up with.

Packing trades a little P2-side work for a large reduction in element count. If you have 32 one-bit logic samples, you can shift them into a single long and send that long as one element. The host unpacks it into 32 separate samples. One element of link traffic now carries what would otherwise have been 32 — a 32× reduction in the number of elements crossing the link for that data.

The savings scale with how much smaller your data is than its container. The denser the packing, the fewer elements you send for the same number of samples.

The packed-data modes

A packed-data mode is named by a **container** (the unit you send — `LONGS_`, `WORDS_`, or `BYTES_`) and a **bit width** (how many bits each unpacked value occupies inside that container). The host divides the container by the bit width to get the count of values it unpacks from each element.

There are **12 modes**. The maximum compression is **32×**, with `LONGS_1BIT`.

Mode	Container bits	Bits per value	Values per element	Compression
LONGS_1BIT	32	1	32	32×
LONGS_2BIT	32	2	16	16×
LONGS_4BIT	32	4	8	8×
LONGS_8BIT	32	8	4	4×
LONGS_16BIT	32	16	2	2×
WORDS_1BIT	16	1	16	16×
WORDS_2BIT	16	2	8	8×
WORDS_4BIT	16	4	4	4×
WORDS_8BIT	16	8	2	2×
BYTES_1BIT	8	1	8	8×
BYTES_2BIT	8	2	4	4×
BYTES_4BIT	8	4	2	2×

Each value is extracted **starting from the LSB** of the element. For LONGS_1BIT, bit 0 is the first unpacked value, bit 1 the second, and so on up to bit 31. For LONGS_8BIT, the low byte is the first value, the next byte the second, and so on. The unpacked ranges are:

Bits per value	Unpacked range	Range if SIGNED
1	0..1	-1..0
2	0..3	-2..1
4	0..15	-8..7
8	0..255	-128..127
16	0..65,535	-32,768..32,767

The ALT and SIGNED modifiers

A mode keyword may be followed by either or both of two optional keywords:

- SIGNED — the host sign-extends each unpacked value. Without it, values are unsigned (the left column above); with it, they take the right column's signed range. Use it when your packed fields represent signed quantities.
- ALT — the host swaps **adjacent same-width fields** throughout the element: neighbouring bits (0 1, 2 3, …), or 2-bit pairs, or nibbles, depending on the mode's field width — a butterfly swap of neighbours across the whole long, not a within-byte or end-to-end reversal. This helps when your source data has its sub-field order swapped from what the display expects — most often bitmap data composed in a standard pixel format.

```
' two signed 16-bit values per long
DEBUG(`SCOPE Sig SIZE 256 128 'val' LONGS_16BIT SIGNED)
```

How to send packed data

Packing is set on the window's creation line — you add the mode keyword to the same DEBUG statement that declares the window. From then on, every element you feed that window is treated as a packed container and unpacked according to the mode. You do the packing on the P2 side; the host does the unpacking.

This example feeds a two-channel LOGIC window with LONGS_1BIT. Each long carries 32 one-bit samples; with two channels declared, the host unpacks the first long as 32 samples of channel 0 and the next long as 32 samples of channel 1. The data is generated in software with the random-number generator, so it runs on a bare board with no wiring:

```
CON _clkfreq = 200_000_000

PUB main() | packed, i
  DEBUG(`LOGIC Stream SAMPLES 256 'D0' 'D1' LONGS_1BIT)
  repeat
    packed := 0
    repeat i from 0 to 31
      ' pack 32 one-bit samples into a long
      packed := (packed << 1) | (GETRND() & 1)
    DEBUG(`Stream `(packed))  ' send one long = 32 samples
    waitms(50)
```

The packing loop builds the long bit by bit. You can build it any way you like — from a streamer capture in hub RAM, from CORDIC results, from a shift register — as long as the bits you want unpacked first land in the low end of the element.

A scope works the same way. Here four 8-bit samples ride in each long under LONGS_8BIT, packed low byte first:

```
CON _clkfreq = 200_000_000

PUB main() | packed, i, ch
  DEBUG(`SCOPE Sig SIZE 256 128 'A' 'B' LONGS_8BIT)
  ch := 0
  repeat
```

continues on next page →

↪ continued from previous page

```
    packed := 0
    repeat i from 0 to 3
      ' four 8-bit values, low byte first
      packed := packed | ((ch++ & $FF) << (i * 8))
    DEBUG(`Sig `(packed))  ' one long = 4 samples
    waitms(20)
```

A BITMAP window unpacks the same formats into pixels. With a `LUT2` (two-bit) color mode you would pack with `LONGS_2BIT`; with a one-bit source you can drive a two-color image using `LONGS_1BIT`, sending one long per 32-pixel row segment:

```
CON _clkfreq = 200_000_000

PUB main() | row, x, packed, bit
  DEBUG(`BITMAP Frame SIZE 32 16 DOTSIZE 8 LUT2 LONGS_1BIT)
  repeat
    repeat row from 0 to 15
      packed := 0
      repeat x from 0 to 31
        bit := ((x + row) & 3) == 0  ' a diagonal stripe pattern
        packed := packed | (bit << x)
      DEBUG(`Frame `(packed))  ' one long = 32 pixels of one row
    waitms(200)
```

Choosing a format

Match the bit width to the size of your values, then pick the container that holds the most of them.

1. **Match the bit width to your data.** One-bit logic channels → a `_1BIT` mode. Values that fit in a nibble (0–15, or −8..7 signed) → a `_4BIT` mode. Byte-sized samples → an `_8BIT` mode. Don't pad small values into a wider field; that throws away the compression.
2. **Pick the widest container you can fill.** For a given bit width, `LONGS_` packs the most values per element, then `WORDS_`, then `BYTES_`. Use `LONGS_` unless your data naturally arrives as words or bytes and repacking into longs would cost more than it saves.
3. **Add `SIGNED` if the values are signed**, and `ALT` only if your sub-byte field order is reversed relative to the display.

So a single-bit logic capture at the highest rate uses `LONGS_1BIT` (32×). A scope sampling a signed 16-bit ADC-style value uses `LONGS_16BIT SIGNED` (2×). A four-level (2-bit) bitmap uses `LONGS_2BIT` (16×).

Where you'd use this

Packing is not a window; it is the **headroom mechanism** the rest of the manual leans on. You reach for it the moment a window cannot keep up with the data — when a fast sample stream would saturate the 2 Mbaud link (Chapter 1).

The concrete case is a **high-rate burst** you have captured in a tight PASM loop — a triggered scope frame, a logic-analyzer capture, a block destined for the FFT — and now have to move over the slow link. Sending one long per sample wastes three or more bytes on values only a few bits wide; packing them (up to 32 samples per long) is what lets the dump **FIT**. This is the readout half of the **capture-and-dump** strategy (Chapter 7): capture fast, pack tight, dump once.

Bandwidth fit: packing multiplies how much *fits*, not how fast the link runs — it buys headroom, not an order of magnitude. When even packed data outruns the link, capture a finite burst and dump it rather than trying to stream live.

Considerations

- **Maximum compression is 32×.** `LONGS_1BIT` is the densest mode. There is no format denser than one bit per value, and no run-length, delta, or general compression scheme — packing is fixed-width bit-field extraction, nothing more.
- **You pack; the host unpacks.** The mode keyword only tells the host how to take the element apart. Building the packed container correctly — right bit width, LSB first — is your code's responsibility.
- **LSB-first ordering is fixed.** The first unpacked value always comes from the low end of the element. Shift your first sample into the low bits. Use `ALT` only to flip sub-byte ordering within each byte, not to reverse whole elements.
- **Packing is per window, set at creation.** All elements fed to that window are unpacked the same way for its lifetime; there is no per-element mode switch.
- **Send whole multiples of the values-per-element count.** A `LONGS_1BIT` LOGIC feed advances 32 samples per element; size your buffers and `SAMPLES` count in multiples of the values-per-element so sets land on element boundaries.
- **The link is still the limit.** Packing reduces *element count*, not the link's raw rate. It is the lever you reach for when a window can't keep up — but the ceiling is still the 2 Mbaud **DEBUG** link.

The windows that read packed data are LOGIC (Chapter 6), SCOPE (Chapter 7), SCOPE_XY (Chapter 8), FFT (Chapter 9), SPECTRO (Chapter 10), and BITMAP (Chapter 4). Each chapter shows the mode keyword in its creation-line table; this chapter is the shared reference for what those keywords mean.

Try it

Start with the LOGIC example above. Change `LONGS_1BIT` to `LONGS_2BIT` and pack two bits per value instead of one — now each long carries 16 two-bit samples (16×), and the unpacked values range 0..3. Then declare the window with `SIGNED` and watch the same bit patterns reinterpret as −2..1. Finally, switch the container from `LONGS_` to `WORDS_` and `BYTES_` for the same bit width and observe how the values-per-element count — and therefore the number of elements you send per screen — changes with the container size.

Chapter 14: Multiple Windows and PASM Debugging

Up to this point each chapter has driven a single window. Real debugging rarely stays that simple: you want a SCOPE showing a waveform *and* a TERM panel printing the numbers behind it, both live at once. And not all of your code is Spin2 — when the part you are chasing runs as PASM2 in its own cog, you need the same `DEBUG` windows from inside that assembly.

This chapter covers both. Neither needs a new mechanism. You already know how to create a window by name and feed it; running several at once is just doing that several times, and debugging from PASM is the same `DEBUG` syntax in a different language. The one thing to unlearn is the idea that windows talk to each other — they do not.

Several windows at once

You can have up to **32 graphical DEBUG displays** open simultaneously. You create each one exactly as you would on its own: a `DEBUG` statement whose backtick names the window type, then a **unique name** you choose. From then on you address each window by its name, independently of every other window.

```
DEBUG(`SCOPE Wave POS 0 0 SIZE 400 200 'Sine' -1000 1000)
DEBUG(`TERM Status POS 420 0 SIZE 40 10)
```

Two windows now exist — a SCOPE named `Wave` and a TERM named `Status`. They are separate windows on the host. The only rule is that each name must be unique, since the name is how every later feed is routed to the right window.

Placing windows with POS

Every display type takes a `POS left top` keyword on its creation line, giving the window's position on the host screen in pixels (default `0, 0`). With more than one window open, set `POS` on each so they do not stack on top of each other. In the example above, `Wave` sits at the top-left corner and `Status` sits 420 pixels to its right — clear of a 400-pixel-wide SCOPE.

> Two host-wide offsets shift *all* displays together: the `DEBUG_DISPLAY_LEFT` and `DEBUG_DISPLAY_TOP` symbols add to every window's `POS` coordinates. Set them in a `CON` block when you want to nudge the whole arrangement without editing each `POS`. They default to `0`.

If you declare several windows **without** `POS`, `pnut_term_ts` places them for you — it offsets each new window from the base display position rather than opening them all on top of each other. That is enough to get started, but the arrangement is automatic, not one you chose. To capture a layout

you *do* like, **drag a window**: while you move it, its title bar shows the window's current `left,top` in pixels. Read those numbers off and encode them into `POS` on that window's creation line, and your chosen arrangement reappears on every run.

Feeding each window in your loop

Once the windows exist, you feed them by name, one statement at a time. A loop that drives both is just both feeds in sequence:

```
CON
  _clkfreq = 200_000_000

PUB main() | ang, sine, count
  ' Two independent windows, each created by name and placed with POS.
  DEBUG(`SCOPE Wave POS 0 0 SIZE 400 200 'Sine' -1000 1000)
  DEBUG(`TERM Status POS 420 0 SIZE 40 10)

  ang := 0
  count := 0
  repeat
    sine := qsin(1000, ang, 256)         ' CORDIC sine, software-generated

    DEBUG(`Wave `(sine))                 ' feed the SCOPE by its name
    DEBUG(`Status 0 "Sample " `udec_(count) 13 "Value:  " `sdec_(sine) 13)

    ang += 4
    count += 1
    waitms(10)
```

This opens a scrolling oscilloscope trace alongside a text panel that reprints the sample count and the current value on every pass. Each `DEBUG` statement names its target; nothing about the SCOPE feed affects the TERM feed or vice versa.

Coordinating windows is just your code

There is **no cross-window *interaction***. Nothing you send to one window changes what another shows, and there is no wildcard "all windows" target, no synchronization group, no shared timestamp, no overlay or picture-in-picture between windows. What you *can* do is address several windows by name in a single feed (below); beyond that, the parser routes each backtick statement to the window names it carries, and that is the whole model.

What looks like coordination is simply your program feeding related data to several windows in the same loop. If you want the SCOPE and the TERM to show the same moment, you send to both

in the same iteration — as the example above does. The "synchronization" is the structure of your loop, not a feature of the windows.

Sending one piece of data to several windows *does* have a built-in shortcut: list more than one instance name after the backtick, and the same elements go to all of them in a single statement. This works when the windows interpret the data the same way — typically windows of the same type, or a shared directive such as `CLEAR` or `SAVE`:

```
DEBUG(`ScopeA ScopeB `(sample))     ' same sample to both SCOPEs
DEBUG(`ScopeA ScopeB CLEAR)         ' one CLEAR clears both
```

When the windows need *different* data — a raw sample to a SCOPE and a formatted line to a TERM — there is no fan-out; you write each feed yourself, in the same loop iteration so they show the same moment:

```
DEBUG(`Wave `(sine))                    ' the SCOPE gets the sample
' the TERM gets the same value, formatted
DEBUG(`Status "now: " `sdec_(sine) 13)
```

> **What does not exist.** There is no `TIMESTAMP`, `OVERLAY`, `ALL_WINDOWS`, `SYNC_GROUP`, `TRIGGER EXTERNAL`, or broadcast command, and no command that makes one window transparent over another. If you need timestamps in a log, format `GETCT()` yourself into a TERM feed; if you need two signals compared, put them on two channels of one SCOPE (Chapter 7) or use SCOPE_XY (Chapter 8). Coordination lives in your code.
>
> A separate, application-wide timestamp facility does exist: defining the `DEBUG_TIMESTAMP` symbol stamps every plain `DEBUG` *message* with the 64-bit CT value. That is a property of the message stream, set once in a `CON` block — not a command you send to a display window.

Debugging from PASM

The `DEBUG` statement is available in PASM2 as well as Spin2. Inside cog assembly — whether a standalone `DAT` program started with `COGINIT`, or an inline `ORG`/`END` block inside a method — you write `DEBUG(...)` with the same backtick display syntax and the same output formatters you use in Spin2. The display windows do not know or care which language fed them.

The difference is what the formatters read. In PASM, the values you display are **cog registers** and immediates, named with PASM syntax. A register name feeds that register's contents; the value forms (`` `() ``, `` `$() ``, `` `udec_() ``, and so on) work exactly as in Spin2.

Here a PASM program running in its own cog drives a SCOPE window, feeding it the value of a cog register:

```
CON
  _clkfreq = 200_000_000

PUB main()
  COGINIT(COGEXEC_NEW, @blink, 0)  ' launch the PASM program in its own cog
  repeat                                ' keep the Spin2 cog alive

DAT
              ORG
blink
              DEBUG(`SCOPE Wave SIZE 400 200 'Ramp' 0 255)
.loop
              ADD       value, #4        ' advance a software ramp
              AND       value, #$FF
              ' feed the window with the register's value
              DEBUG(`Wave `(value))
              WAITX     ##2_000_000
              JMP       #.loop

value         LONG      0
```

The cog creates the `Wave` window with its first `DEBUG`, then feeds it one value per loop. The window opens and animates identically to a Spin2-driven one.

Feeding a TERM from PASM works the same way. This cog reprints a register's value as hex on a text panel:

```
CON
  _clkfreq = 200_000_000

PUB main()
  COGINIT(COGEXEC_NEW, @counter, 0)
  repeat

DAT
              ORG
counter
              DEBUG(`TERM Mon SIZE 30 8)
.loop
              ADD       n, #1
              DEBUG(`Mon 0 "count = " `uhex_long_(n) 13)
```

continues on next page →

↪ continued from previous page

```
15                WAITX     ##50_000_000
16                JMP       #.loop
17
18 n              LONG      0
```

The same DEBUG also works in an **inline** ORG/END block inside a Spin2 method, where the assembly shares the method's local variables:

```
1  CON
2    _clkfreq = 200_000_000
3
4  PUB main() | x
5    DEBUG(`TERM Inline SIZE 30 6)
6    x := 0
7    repeat
8      ORG
9                 ADD       x, #1
10                DEBUG(`Inline 0 "x = " `udec_(x) 13)
11     end
12     waitms(500)
```

Pointers differ between Spin2 and PASM. The interactive commands PC_KEY and PC_MOUSE (Chapter 12) take a pointer to a result buffer. In Spin2 that buffer is in hub, passed as @key; in PASM it must be a **cog register**, passed as #key. This is the one place the language changes the call, and it applies only to those input commands, not to the value formatters.

Considerations

- **The debug link is shared by every window and every cog.** All DEBUG output — from all eight cogs, to all your windows — travels over one serial link, and the cogs time-share it through a hardware lock (LOCK[15]) during their **DEBUG** interrupts. There is no separate channel per window or per cog; everything is serialized onto the same wire and demultiplexed on the host by window name.
- **One shared link means you must pace your output.** Because every feed competes for that one link, the total rate across all windows and cogs is what matters, not the rate of any single window. Messages can stream at up to roughly 10,000 per second before the link saturates. With several windows updating in one loop, keep the loop's waitms/WAITX generous enough that the combined traffic fits.
- **Slow a busy cog with DLY.** Adding `DLY (in Spin2) or DLY(#ms) (in PASM, where the millisecond count is an immediate) as the *last* item in a DEBUG statement delays that cog and

releases the lock, letting other cogs get their messages onto the link. Use it when one cog would otherwise monopolize the output.

- **For high-rate data, pack it rather than sending faster.** When a single window needs more samples than the link comfortably carries one-at-a-time, the packed-data formats (Chapter 13) move many samples per `DEBUG` packet, which is the right tool before you reach for a faster loop.
- **There is no chip-side screenshot or export.** No `DEBUG` command captures the screen or exports a window from the chip. To save a single window's image, send that window's `` `SAVE `` command, which writes a `.bmp` on the host; capturing the whole screen is an action you take on the PC, outside the DEBUG system.
- **Use `` `SAVE `` to capture a window for documentation or a bug report.** Send `` `SAVE 'name' `` at the moment the display shows what you want to keep — for example after a trigger fires or an anomaly appears — and the host writes that frame to a file you can attach to notes or a report. It is the supported way to turn a live window into a static artifact.
- **Keep PASM `DEBUG` out of tight interrupt service routines.** A `DEBUG` taken inside an ISR can skew the cog's timing enough to disturb retriggering. Prefer doing `DEBUG` from cogs that are not running background ISRs (see the Spin2 v5.1 documentation's "DEBUG and interrupts" note).

Try it

Start from the two-window Spin2 example. Add a running peak: track the largest magnitude the signal reaches and print it on the TERM alongside the current value, so the SCOPE shows the waveform while the panel reports its numbers. The complete program below compiles with `pnut_ts` and runs on a bare P2 board with `pnut_term_ts` open — no wiring.

```
1  CON
2    _clkfreq = 200_000_000
3
4  PUB main() | ang, signal, peak, count
5    ' A SCOPE on the left, a TERM status panel on the right.
6    ' Both are created up front, each by its own name, each placed with POS.
7    DEBUG(`SCOPE Trace POS 0 0 SIZE 400 220 SAMPLES 256 'Signal' -1000 1000)
8    DEBUG(`TERM Panel POS 420 0 SIZE 32 8)
9
10   ang   := 0
11   peak  := 0
12   count := 0
13
14   repeat
```

continues on next page →

↪ *continued from previous page*

```
    signal := qsin(1000, ang, 256)         ' software-generated waveform
    if ABS signal > peak
      peak := ABS signal                   ' track the running peak

    ' Coordination is nothing more than feeding both windows
    ' in the same loop:
    DEBUG(`Trace `(signal))                ' one sample to the SCOPE
    DEBUG(`Panel 0 "Samples: " `udec_(count) 13 ...
          "Current: " `sdec_(signal) 13 ...
          "Peak:    " `udec_(peak) 13)   ' a fresh status block to the TERM

    ang   += 4
    count += 1
    waitms(10)
```

Then move the work into a cog: start a PASM program with COGINIT that generates the ramp and feeds the SCOPE itself, and leave the Spin2 cog to print status to the TERM. You will be driving two windows from two cogs over the one shared link — and the only "coordination" anywhere is that both cogs are feeding windows you named.

Chapter 15: Building Control and Status Panels

From Text Dashboards to Sprite-Blitted Instruments

The window chapters each documented one window in isolation. This chapter puts three of them together for a single, practical job: building **status panels** — instrument displays you watch — and **control panels** — on-screen surfaces you operate with the mouse and keyboard.

You have already met every piece. The TERM window positions text at fixed rows and columns (Chapter 3); the PLOT window draws primitives and composites bitmap layers (Chapter 5); `PC_KEY` and `PC_MOUSE` read the host keyboard and mouse back into your program (Chapter 12). A panel is what you get when you combine them.

The reason this takes so little code is that the display does the drawing. Your artwork — a text layout, or a set of bitmap images — is **data**, authored once. Each frame, the P2 issues a short sequence of *position* and *copy* commands; it never rasterizes a glyph or shades a pixel itself. Buffered (double-buffered) mode presents each finished frame at once, so the display does not flicker. And because the same window reads input, a picture becomes a control surface with no added hardware. A working panel is a few dozen lines of Spin2.

This chapter covers two jobs. For **status displays** there is a spectrum of three techniques — positioned text, vector drawing, and sprite-sheet blitting — and the guidance is to use the lightest one that does the job. For **control panels** there is one technique: read input, decide what was hit, act, and redraw. The closing section is a decision guide.

A status panel is positioned output, refreshed in place

A scrolling log answers "what just happened." A status panel answers "what is the state right now" — a fixed face whose labels never move and whose values update in place. Every status technique in this chapter is the same idea at a different cost: draw the static frame once, then overwrite only the parts that change, and never let anything scroll.

Technique 1 — Text dashboards (TERM), the lightest panel

The TERM window already does everything a textual status panel needs, with no artwork at all. You draw the fixed labels once, then position the cursor with the `3` (set row) and `2` (set column) command codes and overprint each value field where it sits. Chapter 3 develops this fully under "A positioned dashboard"; the technique in brief:

```
1 CON _clkfreq = 200_000_000
2
```

continues on next page →

↪ continued from previous page

```
PUB main() | angle, rpm, temp, volts
  DEBUG(`TERM Status SIZE 32 6 TITLE 'System Status')

  ' Draw the static labels once.
  DEBUG(`Status 0 4 "SYSTEM STATUS")       ' clear, pair 0, title at (0,0)
  DEBUG(`Status 3 2 2 0 "RPM  :")          ' row 2, col 0
  DEBUG(`Status 3 3 2 0 "Temp :")
  DEBUG(`Status 3 4 2 0 "Volts:")

  angle := 0
  repeat
    rpm   := 1500 + qsin(500, angle, 360)  ' software-generated readings
    temp  := 40 + qsin(8, angle, 360)
    volts := 120 + qsin(3, angle, 360)

    ' Overprint only the value fields, each at a fixed (row, col). Trailing
    ' spaces pad to a fixed width so a shorter value erases a longer one.
    DEBUG(`Status 3 2 2 8 `udec_(rpm) "   ")
    DEBUG(`Status 3 3 2 8 `udec_(temp) "   ")
    DEBUG(`Status 3 4 2 8 `udec_(volts) "   ")

    angle += 3
    waitms(100)
```

The labels are written once; the loop touches only the three value cells, so the panel reads like a fixed instrument face. Pad every in-place field to a constant width with trailing spaces — overprinting replaces only the characters you send, so without padding, printing `99` over `100` leaves `990`.

Reach for a TERM dashboard whenever the status is textual or tabular: register dumps, state names, counters, a handful of named readings. It needs no images, no layers, and no graphics window. When a value is better shown as a *shape* — a needle, a bar, a trace — move up to PLOT.

Technique 2 — Vector instruments (PLOT), drawn each frame

When a reading is continuous, draw it. The PLOT window's primitives (`LINE`, `CIRCLE`, `BOX`, and the rest) compose an instrument from geometry, and in buffered mode you redraw the whole face each frame without flicker. Chapter 5 builds a complete analog gauge this way under "A worked instrument": center the origin, switch to polar so the needle is a single `LINE` from the center to the reading's angle, and draw the dial with `LINE` and `CIRCLE`.

Vector drawing suits anything geometric and continuously variable — a needle, a moving bar, a level, a live trace — because the shape is computed from the value rather than enumerated. It costs a handful of primitive commands per frame and needs no external artwork. When the face you

want is richer than geometry — a photo-realistic bezel, styled digits, a lamp that lights — move up to sprite-sheet blitting.

Technique 3 — Sprite-sheet panels (PLOT layers), composed from bitmap art

The richest panels are assembled from pre-drawn bitmap **artwork** rather than drawn on the P2. You author the look as Windows BMP files, load them once into the PLOT window's eight off-screen **layers**, and build each frame by **copying rectangles** out of those layers onto the canvas with `CROP`. The P2 issues only copy commands and a little arithmetic; all the pixels were drawn in an image editor. This is the sprite-sheet (blitting) model — the technique behind the richest **DEBUG** instrument panels.

> **Requires `{Spin2_v50}`.** The `LAYER`, `CROP`, and sprite commands are V50 additions. The source file's first line must be `{Spin2_v50}` (or later), compiled with a Spin2 v50+ `pnut_ts`. Without it, these commands are not recognized.

The BMP format is specific. `LAYER` accepts a **24-bit, uncompressed (BI_RGB), no-alpha** Windows BMP — one BMP pixel maps to one canvas pixel, with no scaling. Author each image at the exact device size you will display it. (A short Python + Pillow generator can produce these images and is the practical way to author the artwork; that tooling is documented separately from this manual.)

`CROP` is an opaque copy, and that drives the whole discipline. Because the bitmaps carry no transparency, every `CROP` overwrites its destination rectangle completely. Three consequences follow, and they are the rules of the technique:

- **There is no transparent overlay.** Each sprite cell must already contain the correct background around its shape, because blitting it replaces everything in that rectangle.
- **You erase by restoring, not by clearing.** To remove something, copy the pristine background back over it. So the background layer must contain the empty look of every region you will later overwrite.
- **Seams must match.** The pixels just outside a blitted cell come from the background; the pixels inside come from the sprite. For an invisible seam, author the cell's border to match the background exactly — same colors, same box.

`CROP` has three forms (Chapter 5 documents them in full), and each is one idiom of this technique:

Update Directive

```
CROP layer                            ' whole layer: paint/reset the scene
CROP layer left top width height      ' same spot: erase by restore
CROP layer left top width height x y  ' to (x,y): blit a sprite
```

There are two ways to pack visual state into the artwork. Most panels use both:

- **Whole-state layers** — one full-canvas BMP per state. Load `leds_off.bmp` and `leds_on.bmp` into two layers; to set an indicator, copy its patch from whichever layer holds the wanted state. Strength: trivial code and pixel-perfect alignment (source and destination coincide). Cost: one full image per state, so reserve it for a few states (on/off, up/down).
- **Font / atlas strips** — one BMP holds many equal cells in a row, and you select the cell by arithmetic: `srcX = index * cellWidth`. This is how every numeric readout works, and it yields unlimited values from one small image.

A layer may be larger than the window; the area below the visible canvas is free off-screen storage. One analog-meter design stacks five color copies of its digit font below the gauge in the *same* layer and selects color by source Y — the window shows only the top, and the rest is a palette the code samples from.

The example loads a background and a digit-font strip, then shows a live 3-digit reading by blitting one glyph per column (the font-strip pattern), erasing the box first by restoring background:

```
{Spin2_v50}
CON _clkfreq = 200_000_000

PUB main() | angle, reading
  ' Two BMP layers, loaded once and reused every frame.
  DEBUG(`PLOT Panel SIZE 200 96 POS 200 200 HIDEXY UPDATE)
  DEBUG(`Panel LAYER 1 'panel_bg.bmp')      ' background: frame + empty box
  DEBUG(`Panel LAYER 2 'digits.bmp')        ' font strip: 0-9, each 30x48
  DEBUG(`Panel CROP 1)                      ' paint the background once
  DEBUG(`Panel UPDATE)

  angle := 0
  repeat
    reading := 50 + qsin(50, angle, 360)    ' software reading 0..100
    show3(reading)
    DEBUG(`Panel UPDATE)                    ' one flip per frame
    angle += 4
    waitms(50)

' Blit a fixed-width 3-digit reading using the font strip (form c), after
' erasing the readout box by restoring its background (form b).
PRI show3(value) | d, col, x, div
  DEBUG(`Panel CROP 1 45 24 110 48)         ' erase box: restore bg patch
  div := 100
  repeat col from 0 to 2
    d := (value / div) // 10                 ' this column's digit, 0..9
```

continues on next page →

↪ continued from previous page

```
    x := 50 + col * 36                          ' on-screen slot
    DEBUG(`Panel CROP 2 `(d * 30, 0, 30, 48, x, 28))
    div /= 10
```

Each digit costs the P2 one rectangle copy and a divide — there is no glyph renderer. The same pattern scales to any readout: more columns, a sign cell at the end of the strip, or a second strip for hex (`0`–`9` `A`–`F`). Combine the techniques freely — the analog meter draws its needle with a vector `LINE` over a blitted bezel, then restores a clean hub patch over the needle's base to hide the pivot.

Control panels — reading the surface back

A status panel becomes a control panel when the window reads input. The same `PC_KEY` / `PC_MOUSE` commands from Chapter 12 turn any window into a surface you operate: the user clicks a drawn button or presses a key, and your program acts. Three rules carry over from that chapter — each input command must be the **last** command in its `DEBUG()` statement, the window must have **focus**, and `PC_MOUSE` fills **seven consecutive longs** (`xpos, ypos, wheel, left, middle, right, pixel`, buttons `0`/`-1`, coordinates negative when the pointer is outside).

Two patterns make a panel interactive:

- **Hit-testing** maps an input position to a control. For a rectangular zone, a bounding-box **TEST** (`x` within `x1..x2` and `y` within `y1..y2`) answers "was this control clicked." A row of controls is a `case` on the coordinate.
- **The dirty flag** keeps the panel cheap. Poll input every pass, but recompose and `UPDATE` only when something actually changed. A panel that redraws only on a real event uses almost no link bandwidth while idle.

This panel draws two buttons and a value bar, and adjusts the value from either the arrow keys or a mouse click on a button. It recomposes only on a change:

```
{Spin2_v50}
CON _clkfreq = 200_000_000

PUB main() | m[7], key, value, dirty, lastL
  DEBUG(`PLOT Ctrl SIZE 300 140 POS 200 200 BACKCOLOR $202020 HIDEXY UPDATE)
  value := 50
  dirty := true                                 ' force the first draw
  lastL := 0

  repeat
    ' Poll both inputs; each is the LAST command in its statement.
    key := 0
```

continues on next page →

↪ continued from previous page

```
    DEBUG(`Ctrl PC_KEY(@key))
    DEBUG(`Ctrl PC_MOUSE(@m))

    case key                                  ' keyboard: arrows nudge
      1: value -= 5                            ' Left  arrow
         dirty := true
      2: value += 5                            ' Right arrow
         dirty := true

    if m[3] AND NOT lastL                      ' left-button press edge
      if in_box(m[0], m[1], 25, 45, 75, 95)
        value -= 5
        dirty := true
      if in_box(m[0], m[1], 225, 45, 275, 95)
        value += 5
        dirty := true
    lastL := m[3]

    value := 0 #> value <# 100                 ' clamp to range

    if dirty
      draw(value)
      DEBUG(`Ctrl UPDATE)
      dirty := false
    waitms(20)

' Bounding-box hit test: true when (x,y) is inside the rectangle.
PRI in_box(x, y, x1, y1, x2, y2) : hit
  hit := (x >= x1) AND (x <= x2) AND (y >= y1) AND (y <= y2)

PRI draw(v) | len
  DEBUG(`Ctrl CLEAR)
  DEBUG(`Ctrl COLOR $C04040 SET 50 70 BOX 50 50 0 255)   ' minus button
  DEBUG(`Ctrl COLOR $40C040 SET 250 70 BOX 50 50 0 255)  ' plus button
  len := 1 + v * 150 / 100                   ' value 0..100 -> bar length
  DEBUG(`Ctrl COLOR $4080FF)
  DEBUG(`Ctrl SET `(75 + len / 2, 70) BOX `(len, 24, 0, 255))
```

Click the window to give it focus, then click a button or use the arrow keys. The `lastL` variable holds the previous left-button state so a click fires once on the press edge rather than every poll

while the button is held. The `dirty` flag means the panel sits idle — polling but not redrawing — until an input changes the value.

Where you'd use this

In computer science and computer engineering, the panel techniques are for building **live dashboards and operator consoles** — a composed instrument face that reads a system at a glance — and **interactive control and tuning panels**, the human-in-the-loop surfaces of a bring-up rig.

On an embedded project, you reach for them to build a one-screen status console for a running machine, a tuning panel with on-screen buttons and live readouts, or a hardware-in-the-loop **TEST** rig you drive while the firmware runs.

Bandwidth fit: a panel of readouts and controls updates at human rates and fits the link easily; only a panel fed by a fast *sample stream* needs the packed feeds noted below.

Extension (real hardware): wire the panel's readouts to real sensor values and its buttons to real outputs, and the same composed face becomes the machine's actual operator interface.

Considerations — choosing a technique

- **Use the lightest technique that does the job.** Textual or tabular status → a TERM dashboard, no artwork. A continuous, geometric value → PLOT vector drawing. A rich or photo-realistic face → PLOT sprite-sheet blitting. Do not author BMP layers for a job a positioned text field handles.
- **Always run panels in buffered mode.** Create the window with `UPDATE`, compose the whole frame, then issue one `` `UPDATE ``. Batching every change behind a single flip is what makes the panel flicker-free.
- **Erase by restoring, never by guessing.** In the blitting technique there is no clear-to-blank; copy clean background over a region (`CROP` form b) or repaint the whole scene (form a). The background layer must hold the empty look of everything you overwrite.
- **Gate redraws behind a dirty flag** for interactive panels. Poll input continuously, but recompose only on a real change so an idle panel costs almost nothing.
- **High-rate panels need packed feeds.** When a panel is driven by a fast sample stream, the debug link is the bottleneck; pack many samples per `DEBUG()` **CALL** (Chapter 13).
- **Run several panels at once.** Each `DEBUG(`\PLOT …)`OR`**DEBUG**(\XPROTECTINLINECODE2X with a distinct name is an independent window; a cog can drive a wall of panels, and PASM2 can drive them too (Chapter 14).

Try it

Start from the TERM dashboard and add one interactive field: poll `PC_KEY` in the loop and let the `+` and `-` keys adjust one of the readings, redrawing only on a keypress. Then rebuild the same panel in PLOT with a sprite-digit readout (Technique 3) and a mouse-clickable button (the control

pattern) — you will have moved one panel across the whole spectrum, from positioned text to a blitted, clickable instrument, using nothing but the debug link and a bare P2 board.

Part IV

Appendices

The appendices are for when you already know what you want and need the exact spelling. Appendix A is the command reference: every window's creation and runtime commands, gathered per window with their ranges and defaults, and the commands shared by all windows listed once. Appendix B is the packed-data format reference — the packing modes of Chapter 13 set side by side with their containers and compression. Appendix C collects the conventions the windows share: the `$RRGGBB` color form and named constants, TERM's color pairs, and the coordinate systems.

Reach here to confirm a parameter, not to meet a window for the first time — the chapters in Parts I through III do the teaching.

Appendix A: Command Reference

A per-window summary of creation/configuration directives and runtime commands. Ranges and defaults are as documented in each window's chapter. Commands shared by every window are listed once at the end.

TERM — text terminal (Chapter 3)

Create: `DEBUG(`TERM Name <config>)`

Configuration directives:

Directive	Notes
`TITLE 'text'`	
`POS left top`	
`SIZE cols rows`	1–256, default 40×20
`TEXTSIZE pts`	
`COLOR`	8 values = 4 fg/bg pairs
`BACKCOLOR rgb`	
`UPDATE`	
`HIDEXY`	

Feed — command codes:

Code	Effect
`0`	clear + home
`1`	home
`2 col`	set column
`3 row`	set row
`4–7`	select color pair 0–3
`8`	backspace (cursor only)
`9`	tab
`10 / 13`	newline
`32–255`	character

Runtime commands:

Directive	Notes
`CLEAR`	
`UPDATE`	buffered repaint
`SAVE`	
`CLOSE`	close + free this window

BITMAP — pixel display (Chapter 4)

Create: `DEBUG(`BITMAP Name <config>)`

Configuration directives:

Directive	Notes
`TITLE`	
`POS`	
`SIZE w h`	1–2048
color mode	see Appendix C
`DOTSIZE`	1–256
`SPARSE`	
`TRACE`	0–15
`RATE`	
`UPDATE`	
`HIDEXY`	

Feed: numeric pixel values (streamed per the trace pattern; packable — Appendix B).

Runtime commands:

Directive	Notes
`LUTCOLORS`	load palette
`TRACE`	
`RATE`	
`SET x y`	
`SCROLL`	
`CLEAR`	
`UPDATE`	
`SAVE`	
`CLOSE`	close + free this window

BITMAP has no drawing primitives — those belong to PLOT.

PLOT — vector drawing canvas (Chapter 5)

Create: `DEBUG(`PLOT Name <config>)`

Configuration directives:

Directive	Notes
`TITLE`	
`POS`	
`SIZE w h`	32–2048, default 256×256
`DOTSIZE x {y}`	1–256
color mode	see Appendix C
`LUTCOLORS`	
`BACKCOLOR`	
`UPDATE`	
`HIDEXY`	

Position / state (Update directives):

Directive	Notes
`SET x y`	
`ORIGIN {x y}`	
`PRECISE`	
`COLOR rgb`	
`OPACITY 0-255`	
`LINESIZE`	
`CARTESIAN {flipy {flipx}}`	
`POLAR {twopi {theta}}`	
`TEXTSIZE`	

Primitives (cursor-relative):

Directive	Notes
`DOT {linesize {opacity}}`	
`LINE x y {linesize {opacity}}`	
`CIRCLE diameter {...}`	
`OVAL w h {...}`	
`BOX w h {...}`	
`OBOX w h xr yr {...}`	
`TEXT {size {style {angle}}} 'string'`	

Layers / sprites:

Directive	Notes
`LAYER n 'file.bmp'`	n = 1–8
`CROP n` / `CROP n AUTO x y` / `CROP n left top w h {x y}`	crop/composite a layer
`SPRITEDEF id xsize ysize ...`	id 0–255, size 1–32
`SPRITE id orient ...`	orient 0–7

Runtime commands:

Directive	Notes
`CLEAR`	
`UPDATE`	buffered repaint trigger
`SAVE`	
`CLOSE`	close + free this window

LOGIC — logic analyzer (Chapter 6)

Create: `DEBUG(`LOGIC Name <config> <channels>)`

Configuration directives:

Directive	Notes
`TITLE`	
`POS`	
`SAMPLES`	4–2047
`SPACING`	1–32
`RATE`	1–2048
`DOTSIZE`	0–32
`LINESIZE`	1–32
`TEXTSIZE`	
`COLOR back grid`	
`HIDEXY`	
packing keyword	see Appendix B

Channels (as creation elements): `'label' {bit-count} {RANGE} {color}`. Up to 32 channels; one shared 2048-sample buffer.

Runtime commands:

Directive	Notes
`TRIGGER mask match {offset}`	
`HOLDOFF`	2–2048
`CLEAR`	
`SAVE`	
`CLOSE`	close + free this window

Shows raw waveforms — no built-in protocol decoding.

SCOPE — time-domain oscilloscope (Chapter 7)

Create: `DEBUG(`SCOPE Name <config> <channels>)`

Configuration directives:

Directive	Notes
`TITLE`	
`POS`	
`SIZE w h`	32–2048
`SAMPLES`	16–2048
`RATE`	1–2048 divisor
`DOTSIZE`	
`LINESIZE`	
`TEXTSIZE`	
`COLOR back grid`	
`HIDEXY`	
packing	see Appendix B

Channels (as creation elements): `'label' {AUTO | lo hi} {tall} {base} {grid} {color}`. Up to 8 channels.

Feed: one numeric value per channel per time step.

Runtime commands:

Directive	Notes
`TRIGGER channel {AUTO | arm fire} {offset}`	rising if fire >= arm, else falling
`HOLDOFF`	
`CLEAR`	
`SAVE`	
`CLOSE`	close + free this window

SCOPE_XY — XY / phase display (Chapter 8)

Create: `DEBUG(`SCOPE_XY Name <config> <channels>)`

Configuration directives:

Directive	Notes
`TITLE`	
`POS`	
`SIZE radius`	single value
`RANGE extent`	single, symmetric ±
`SAMPLES`	persistence depth; 0 = no fade
`RATE`	
`DOTSIZE`	
`TEXTSIZE`	
`COLOR back {grid}`	
`POLAR {twopi {offset}}`	
`LOGSCALE`	
`HIDEXY`	

Channels: `'name' {color}` (up to 8).

Feed: `` `(x, y) `` pairs (in channel order).

Runtime commands:

Directive	Notes
`CLEAR`	
`SAVE`	
`CLOSE`	close + free this window

FFT — frequency spectrum (Chapter 9)

Create: `DEBUG(`FFT Name <config> <channels>)`

Configuration directives:

Directive	Notes
`TITLE`	
`POS`	
`SIZE w h`	32–2048 px
`SAMPLES N {first last}`	N = FFT size, power of 2, 4–2048; optional bin range
`RATE`	1–2048
`DOTSIZE`	
`LINESIZE`	−32…32; negative = filled bars
`TEXTSIZE`	
`COLOR back grid`	
`LOGSCALE`	log2 amplitude
`HIDEXY`	

Channels: `'label' MAG-shift(0-11) high tall base grid color`. A Hanning window is always applied; it is not selectable.

Runtime commands:

Directive	Notes
`CLEAR`	
`SAVE`	
`CLOSE`	close + free this window

SPECTRO — spectrogram / waterfall (Chapter 10)

Create: `DEBUG(`SPECTRO Name <config>)` (single channel)

Configuration directives:

Directive	Notes
`TITLE`	
`POS`	
`SAMPLES`	FFT size, power of 2, 4–2048
`DEPTH`	1–2048
`RANGE ceiling`	single value
`RATE`	1–2048; default SAMPLES/8
`TRACE`	0–15; bit 3 = scroll
`MAG`	0–11
`DOTSIZE`	1–16
`LOGSCALE`	
`HIDEXY`	
color mode	LUMA8/W/X, HSV16/W/X
packing	see Appendix B

Runtime commands:

Directive	Notes
`CLEAR`	
`SAVE`	
`CLOSE`	close + free this window

MIDI — piano-keyboard display (Chapter 11)

Create: `DEBUG(`MIDI Name <config>)`

Configuration directives:

Directive	Notes
`TITLE`	
`POS`	
`SIZE`	key-size multiplier 1–50, default 4
`RANGE first last`	notes 0–127, default 21–108
`CHANNEL`	0–15, default 0
`COLOR white-active black-active`	two RGB24; defaults cyan/magenta

Feed: raw MIDI bytes as numeric values — Note-On `$9n note velocity`, Note-Off `$8n note velocity` (running status supported). Only `$8n`/`$9n` are recognized.

Runtime commands:

Directive	Notes
`CLEAR`	
`SAVE`	
`CLOSE`	close + free this window

Commands common to every window

- `DEBUG(`Name PC_KEY(@keyvar))` — host writes the latest key code (0 if none) into the long at `@keyvar`. See Chapter 12 for the key-code table.
- `DEBUG(`Name PC_MOUSE(@mousevar))` — host fills a 7-long array: xpos, ypos, wheel, left, middle, right (each button 0 or −1), pixel-under-cursor. See Chapter 12.
- `DEBUG(`Name CLEAR)` — clear the window.
- `DEBUG(`Name SAVE {WINDOW} 'file')` — save the window image to a host file.
- `DEBUG(`Name CLOSE)` — close and free this one window. A different action from ending the whole **DEBUG** session; works on all nine window types.

Appendix B: Packed-Data Format Reference

Packed-data modes let one DEBUG **CALL** carry many small samples, multiplying the throughput of the debug link. You name a packing mode in a window's feed, then send packed longs/words/bytes; the window unpacks them. See Chapter 13 for usage.

The twelve formats

Keyword	Container	Values per container	Compression
LONGS_1BIT	long	32	32×
LONGS_2BIT	long	16	16×
LONGS_4BIT	long	8	8×
LONGS_8BIT	long	4	4×
LONGS_16BIT	long	2	2×
WORDS_1BIT	word	16	16×
WORDS_2BIT	word	8	8×
WORDS_4BIT	word	4	4×
WORDS_8BIT	word	2	2×
BYTES_1BIT	byte	8	8×
BYTES_2BIT	byte	4	4×
BYTES_4BIT	byte	2	2×

Maximum compression is **32×** (LONGS_1BIT). Fields are extracted LSB-first.

Modifiers

- SIGNED — sign-extend each field instead of treating it as unsigned. Signed ranges: 1-bit −1…0, 2-bit −2…1, 4-bit −8…7, 8-bit −128…127, 16-bit −32768…32767.
- ALT — reverse the sub-field order within each container.

Syntax: <packing-mode> {ALT} {SIGNED}.

Choosing a format

Match the field width to your data's range: a single digital line packs at `LONGS_1BIT` (32×); an 8-bit sample at `LONGS_8BIT` (4×) or `BYTES_*`. Windows that accept packed data include BITMAP, LOGIC, SCOPE, SCOPE_XY, FFT, and SPECTRO.

Appendix C: Color and Coordinate Reference

Color values

Colors are 24-bit RGB written `$RRGGBB` (for example `$FF7F00` is orange). You can also use the Spin2 named color constants where a color is expected.

TERM color pairs

TERM holds four foreground/background pairs, selected at runtime with command codes `4`–`7`. Defaults:

Pair	Code	Foreground	Background
0	`4`	Orange	Black
1	`5`	Black	Orange
2	`6`	Lime	Black
3	`7`	Black	Lime

Set your own with `COLOR` on the creation line (eight values: fg0 bg0 fg1 bg1 fg2 bg2 fg3 bg3).

BITMAP color modes

BITMAP selects a pixel format with one of these mode keywords (default `RGB24`): `LUT1` `LUT2` `LUT4` `LUT8` · `LUMA8` `LUMA8W` `LUMA8X` · `RGBI8` `RGBI8W` `RGBI8X` · `HSV8` `HSV8W` `HSV8X` · `HSV16` `HSV16W` `HSV16X` · `RGB8` (3:3:2) · `RGB16` (5:6:5) · `RGB24` (8:8:8). `LUT`-mode palettes are loaded with `LUTCOLORS`.

SPECTRO color modes

SPECTRO maps magnitude to color using `LUMA8` / `LUMA8W` / `LUMA8X` (default `LUMA8X`) or `HSV16` / `HSV16W` / `HSV16X`.

Coordinates

Window	Coordinate system
TERM	Character grid, 0-based: column 0…cols−1, row 0…rows−1 (top-left = 0,0)
BITMAP	Pixels from the top-left; the trace cursor advances per the `TRACE` pattern, or you place it with `SET x y`
PLOT	Cartesian by default (origin movable with `ORIGIN`), or `POLAR` (rho, theta); `SET` positions the drawing cursor
SCOPE	Time advances left-to-right one step per sample set; vertical is the channel's value range (`AUTO` or `lo hi`)
SCOPE_XY	Centered; `SIZE` is the radius and `RANGE` the symmetric ± extent; `POLAR` accepts (rho, theta)
FFT	Horizontal = frequency bin (bin k k × sample_rate / N); vertical = magnitude
SPECTRO	Horizontal = frequency; the display scrolls over time per `TRACE`/`RATE`; color = magnitude
MIDI	A piano keyboard spanning the note `RANGE`; key fill height = note velocity

Frequency-in-hertz labeling for FFT and SPECTRO is your own calculation from the sample rate and FFT size — the windows display bins, not hertz.

Index

A chapter-referenced index to the windows, command keywords, and key concepts in this manual. Each entry links to the chapter(s) where the topic is explained.

A

B

C

D

F

H

R

S

T

U

W