

VETUS PARS

Definitive Test-Language Coding Guide

Handwritten code and VetusGUI workflows

Source-of-truth review: current Vetus Pars firmware and current VetusGUI
September 2026

Authoritative for the reviewed code state. Items labeled “Current implementation caution” are real limitations or mismatches found in that state and should be rechecked after firmware or GUI changes.

How to use this guide

This manual describes the code that Vetus Pars actually executes and the source text VetusGUI currently compiles. It is written for both people who type .chip files directly and people who build instructions with the GUI helpers.

- Start with Chapters 1–5 before writing a test. They define instruction rows, values, registers, pin numbering, and safe electrical sequencing.
- Use Chapters 6–10 for address buses, binary side words, flow, data, display, and add-on boards.
- Use the opcode reference as the everyday lookup. Syntax is shown exactly as it is typed.
- “Register value allowed” means an input may be a literal, a DIM constant, or #Rn. “Register slot” means type the destination number, such as 3—not #R3.

Status labels

Label	Meaning
Supported	Present in the VetusGUI compiler and handled by the current firmware.
GUI-assisted	VetusGUI supplies a picker or chip-pin helper; the saved text is still normal Vetus code.
Current implementation caution	The reviewed GUI and firmware disagree, or the firmware behavior has a material limitation.
Reserved / raw only	A numeric opcode exists in headers or firmware but is not offered as a normal VetusGUI instruction.

Contents

1. Program model and file anatomy
2. Values, constants, registers, and conditions
3. Chip pins and safe sequencing
4. ALLDIR and ALLEVEL binary words
5. Address buses: DEFADDR, SETADDR, GETADDR
6. Eight-bit maps and arrays
7. Flow control and functions
8. Timing, display, images, and strings
9. Add-on board commands and safety
10. Complete worked examples
11. Detailed opcode reference
12. Known implementation cautions and reserved codes

1. Program model and file anatomy

A Vetus test is a line-oriented program. One executable source instruction becomes four 32-bit words: opcode, argument 1, argument 2, and argument 3. That is 16 bytes per instruction. The current firmware buffer holds at most 256 executable instructions (4,096 bytes). Blank lines, comments, and DIM declarations do not consume instruction rows.

What counts as an instruction row

```
DIM DATA_PIN 7          // no instruction row
/* setup section */      // no instruction row
SETOUT DATA_PIN         // instruction 1
SETPIN DATA_PIN 0x1     // instruction 2
DELUS 10                 // instruction 3
```

JMP and CALL use compiled instruction rows, not the visible source-line number. VetusGUI's gutter shows the compiled row numbers and marks invalid rows. If you add or remove executable instructions, recheck literal jump targets.

Source syntax

- Opcodes are case-insensitive in VetusGUI and are normally written uppercase.
- Arguments are separated by whitespace. There are no commas.
- Both // line comments and /* ... */ block comments are recognized by the GUI tokenizer.
- Every normal instruction accepts zero to three arguments, as listed in the reference.
- Unknown text is a compile error. A raw opcode escape uses ~0xNN followed by up to three values.

CURRENT IMPLEMENTATION CAUTION — Raw ~0xNN is a developer escape. This was created in case VetusGUI does not keep up with firmware. If there is an opcode known by firmware but not the GUI, you may use ~0xAD for example and the opcode 0xAD will be sent to Vetus. It bypasses the normal opcode vocabulary and can reach unsupported firmware paths. Use it only when the exact current firmware handler has been reviewed.

A safe test skeleton

```
ASAFE
// Define pins and initial LOW levels before enabling outputs.
ALLEVEL 0b00000000 0b00000000
ALLDIR  0b00000111 0b00000000

// Test body here.

ASAFE
PASSFAIL 1
ENDTEST
```

PASSFAIL 1 produces the green PASSED end screen; PASSFAIL 0 produces FAILED. If the engine records an error, the run is ended through the failure cleanup path.

2. Values, constants, registers, and conditions

Literal values

Form	Example	Meaning
Decimal	25	Unsigned decimal 25
Hexadecimal	0x19	25; useful for colors, masks, and packed pins
Binary	0b00011001	25; useful for ALLDIR/ALLEVEL and masks
Register input	#R3	Use the current 32-bit value stored in register 3
DIM name	DATA_PIN	Replace with the constant declared by DIM

Numbers compile as unsigned 32-bit values. Hex and binary are especially helpful when individual bits matter. Negative source literals are not part of the normal VetusGUI value grammar.

DIM constants

```
DIM DATA_PIN 7
DIM WRITE_LEVEL 0b1
DIM START_VALUE 0x20
```

- A name must start with a letter or underscore; remaining characters may be letters, digits, or underscores.
- Names are case-insensitive, unique, and cannot be DIM or an opcode name.
- The value must directly be decimal, hexadecimal, binary. A DIM cannot be defined from another DIM.
- DIM creates no runtime storage and no instruction row. A register-valued DIM still resolves at runtime because it compiles as a register reference.

Registers

The engine has R0–R31, each 32 bits. Use #Rn only when reading the contents of a register. For a destination, type the bare slot number.

```
REGL0D 3 0x20      // put 0x20 into R3
ADD #R3 5 4         // R4 = R3 Content + 5
RDPIN 7 5           // read chip pin 7 into R5
SHOSTR 0 4 20       // String 0 may format #R4 and #R5
```

Comparison codes

Code	Operator	True when
0 / EQ	==	left equals right
1 / NE	!=	left differs from right
2 / LT	<	left is less than right (unsigned)
3 / GT	>	left is greater than right (unsigned)
4 / LE	<=	left is less than or equal to right
5 / GE	>=	left is greater than or equal to right
6 / DIV2	even	left modulo 2 is zero; the right value is ignored

VetusGUI accepts the named operators where its operator picker is used and compiles them to these numeric codes. Handwritten code may use the numeric code.

3. Chip pins and safe sequencing

Pin numbers in Vetus code are normal package pin numbers, starting at 1. For a DIP package, pins 1 through $N/2$ run down Side A. Side B is represented in physical counter-clockwise order: the first Side-B bit is the highest-numbered pin N , then $N-1$, continuing to $N/2+1$.

Logical group	Bit 0	Bit 1	Last used bit
Side A	Pin 1	Pin 2	Pin $N/2$
Side B	Pin N	Pin $N-1$	Pin $N/2+1$

VetusGUI's chip picture uses the selected chip's pin count and inserts the same user pin numbers. A GUI-generated line and a handwritten line therefore have identical meaning.

Direction, level, and high impedance

- Input means the test socket does not intentionally drive the chip pin.
- Output means Vetus may drive the pin to the level selected by SETPIN/ALLEVEL.
- Hi-Z disconnects the tester-side path as far as the current hardware control permits. ASAFE applies Hi-Z to every socket line.
- Set output levels before enabling outputs whenever possible. This avoids a brief wrong-level pulse.
- End every test and every early failure path with ASAFE before ENDTEST whenever your code can do so. This done automatically but redundancy is sometimes helpful.

Recommended startup sequence

```

ASAFE
ALLEVEL 0b00000000 0b00000000 // preload LOW
ALLDIR 0b00101101 0b00000010 // then enable selected outputs

```

Single and multiple pins

SETOUT and SETIN change direction. SETPIN changes level but does not make the pin an output. MULTIO, MULTII, MULTIHI, and MULTILO do the same operation on exactly three pins.

4. ALLDIR and ALLEVEL binary words

ALLDIR sideA sideB changes direction for every pin in the selected package. A 1 means output; a 0 means input. ALLEVEL sideA sideB changes levels. A 1 means HIGH; a 0 means LOW. Only the lowest N/2 bits of each word are used; higher bits are ignored.

Bit order—the rule to remember

Side A counts upward from pin 1. Side B counts downward from the highest pin. Read either binary word from right to left because the rightmost digit is bit 0.

Word bit	Side A pin	Side B pin (N-pin package)
0	1	N
1	2	N-1
2	3	N-2
...	...	...
N/2-1	N/2	N/2+1

14-pin example

For a 14-pin chip, each side uses seven bits. To make pins 1, 3, and 7 outputs on Side A, and pins 14 and 10 outputs on Side B:

```
ALLDIR 0b1000101 0b0010001
// Side A bits 6..0 = pins 7..1
// Side B bits 6..0 = pins 8..14
```

The Side-A word 0b1000101 has bits 6, 2, and 0 set: pins 7, 3, and 1. The Side-B word has bits 4 and 0 set: pin 10 and pin 14.

16-pin example with levels

```
ALLEVEL 0b00000101 0b10000001
ALLDIR 0b00000101 0b10000001
```

Side A drives pins 1 and 3 HIGH. Side B drives pins 16 and 9 HIGH. The same masks are used for direction here, so those four pins become outputs. In real tests, direction and level masks may differ.

Hex equivalents

Binary is easiest to audit, but the words may be decimal, hex, DIM constants, or register values. For eight pins per side, 0b10000001 equals 0x81. For twenty pins per side, keep the word within the lowest 20 bits.

CURRENT IMPLEMENTATION CAUTION — ALLEVEL changes levels even for pins currently configured as input; those levels become important if a later instruction makes the pin an output. Treat it as a preload and audit the subsequent ALLDIR carefully.

5. Address buses: DEFADDR, SETADDR, GETADDR

The 32-bit address map connects logical address bits A0–A31 to arbitrary package pins. DEFADDR defines eight consecutive address bits at a time. Valid bases are 0, 8, 16, and 24.

Packed-pin byte order

Each 32-bit packed argument contains four one-byte pin numbers. The least-significant byte is the first logical address bit. When written as ordinary hex, the byte groups therefore appear visually reversed.

Argument byte	Logical mapping at base B
bits 7–0 (rightmost hex byte)	A(B+0)
bits 15–8	A(B+1)
bits 23–16	A(B+2)
bits 31–24 (leftmost hex byte)	A(B+3)

```
DEFADDR 0 0x08070605 0x0C0B0A09
```

```
// A0=pin5, A1=6, A2=7, A3=8
```

```
// A4=pin9, A5=10, A6=11, A7=12
```

A second group can define A8–A15:

```
DEFADDR 8 0x14131211 0x18171615
```

```
SETADDR 16 0x1234 1
```

SETADDR nbits value direction uses bit 0 of value for A0, bit 1 for A1, and so on. Direction 1 configures mapped pins as outputs and drives the address. Direction 0 configures them as inputs, however, GetAddress converts them to inputs automatically so 0 should never need to be used. Use nbits from 1 through 32 and define every bit you intend to use.

Writing DEFADDR by hand

- List the pins in logical order A0, A1, A2...
- Put the first four pins into the low word in reverse visual hex-byte order. Example pins 5,6,7,8 become 0x08070605.
- Put the next four into the high word the same way.
- A zero byte means unused. Unused zero bytes may trail the defined pins; do not leave a zero gap followed by another pin.
- Do not repeat a pin and do not specify a pin above the selected chip's pin count.

Using VetusGUI

Choose DEFADDR, select base 0/8/16/24, then click package pins in A-bit order. Base refers to the section of a theoretical address. 0 refers to the first 8 bits. If an address is 16 bits you would need to define the first 8 bits using 0 as the base, then the next 8 bits using 8 as the base and so on up to 32 bits. The first selected pin is A0 (or the selected base), the next is the following address bit, and so on. VetusGUI packs the two words automatically and validates duplicate, missing-middle, and out-of-range pins.

Reading an address

```
GETADDR 16 3 // read mapped bus into R3
```

GETADDR places the bus value in the bare destination slot. The current firmware reads from A0 upward and stops at the first unmapped address bit.

SETADDR drives a value onto the mapped bus. GETADDR changes the mapped pins to inputs and reads the bus into a register. GETADDR reads up to the requested number of bits, starting at A0. It stops sooner if it encounters the first undefined address bit. Address definitions must therefore be continuous from A0 through the highest bit you intend to read.

6. Eight-bit maps and arrays

DEF8, SET8, and GET8

DEF8 creates a reusable eight-pin map. Its two packed words use exactly the same low-byte-first pin order as DEFADDR. SET8 sends an 8-bit value through the map; GET8 reads it into a register.

```
DEF8 0 0x08070605 0x0C0B0A09
SET8 0 0xA5 1          // bit0→pin5, bit7→pin12
SET8 0 0 0             // make mapped pins inputs
GET8 0 4               // R4 receives the eight input bits
```

VetusGUI limitation — The current firmware provides six 8-bit address maps, numbered 0 through 5. VetusGUI may currently accept larger map numbers, but they are invalid. Always use map indexes 0–5, 1, 2, 3, 4, or 5.

Arrays A and B

There are two 32-element arrays. Array selector 0 means A; any nonzero selector means B. They are cleared at test-engine reset.

Instruction	Meaning
ARRADD array index value	Write or append. If index 0 and array is empty, write slot 0; if index 0 and nonempty, append; otherwise write the literal/resolved index and extend the length.
ARRGET array index dest	Copy an existing item into a bare register slot. If index is beyond current length, the destination is unchanged.
PIN2ARR array packedLo packedHi	Append all eight packed bytes as array values, including zero bytes, until the array reaches 32 items.

```
ARRADD 0 0 11          // A[0]=11
ARRADD 0 0 22          // append A[1]=22
ARRGET 0 1 2           // R2=22
PIN2ARR 1 0x04030201 0x08070605
```

PIN2ARR is often used to create an ordered pin list. Its byte order is identical to address packing: rightmost byte first.

7. Flow control and functions

Counted loops

```
LOOP 100
    // body
ENDLOOP
```

LOOP repeats its body count times. The engine supports up to 32 active counted/do-until loop records. BRKLOOP exits the nearest enclosing LOOP, WHILE, or DOUNTIL block and understands nesting of the same loop type.

Important — LOOP must use a count of 1 or greater. Never use LOOP 0, because it produces an extremely long loop.

WHILE and DOUNTIL

```
WHILE LT #R0 100
    REGINC 0
+ENDWHL

DOUNTIL
    RDPIN 6 1
ENDUNTIL EQ #R1 1
```

WHILE tests before the body. DOUNTIL always executes the body once and tests at ENDUNTIL. Conditions are unsigned and use the codes in Chapter 2. Nested matching is supported by forward/backward scans, but complex cross-jumps into loop bodies should be avoided.

IF blocks

```
IF EQ #R2 0
    PASSFAIL 0
ENDIF
```

IF executes the body only when the condition is true. Nested IF blocks are skipped correctly.

JMP and GOSTRT

JMP target uses a one-based compiled instruction row. JMP 1 goes to the first executable instruction. GOSTRT resets the program counter to the first instruction. JMP rebuilds loop state for the destination; GOSTRT does not perform a full engine reset.

Functions

```
CALL pulse_write
PASSFAIL 1
ENDTEST

FUNC pulse_write
    SETPIN 4 1
    DELUS 2
    SETPIN 4 0
ENDFUNC
```

- Function names are case-insensitive, unique, and at most 12 UTF-8 bytes.
- FUNC and ENDFUNC are executable rows. Place all functions after ENDTEST so normal execution never falls through into a function.

VETUS PARS | DEFINITIVE TEST-LANGUAGE GUIDE

- CALL is converted by VetusGUI into the one-based instruction row of the FUNC declaration.
- The current firmware stores one return address only. Calls made while already inside a function are ignored; nested and recursive calls are unsupported.

CONTINUED BELOW

8. Timing, display, images, and strings

Delays and clock

DELUS and DELMS are cooperative delays: the engine remains on that instruction until the timer expires. DELMS multiplies the value by 1,000 in 32-bit arithmetic, so avoid extremely large millisecond values. CLKST pin kHz starts the single test clock; CLKSTP stops it. The GUI exposes kilohertz. Current firmware also has a hidden third hertz field, but normal GUI output writes it as zero. The kHz path clamps at 500 kHz.

Display drawing model

Drawing instructions update the display's working buffer. Use UDSCRN to request the DMA refresh. SHOWIMG's own GUI description recommends UDSCRN followed by DELMS 10.

```
FILSCR 0x0000
TXTCOL 0xFFFF 0x0000
SHOSTR 0 4 8
SHOWIMG 11 40 4
UDSCRN
DELMS 10
```

Item	Range / packing
Screen	160 × 128 pixels in current firmware; x is 0–159, y is 0–127.
Colors	RGB565: red bits 15–11, green 10–5, blue 4–0. Common: black 0x0000, white 0xFFFF, red 0xF800, green 0x07E0, blue 0x001F.
Strings	Eight slots, 0–7; storage slot length is 64 bytes including termination/metadata constraints.
Images	SHOWIMG uses zero-based image index; current maximum constant is 240.
SEVSEG arg2	x in upper 16 bits, y in lower 16 bits: (x << 16) y.
SEVSEG arg3	size in bits 16–23, RGB565 color in low 16 bits.

String formatting with registers

Place normal printf-style conversion text in a string slot and place #Rn markers in the same order as its conversions. The firmware removes each #Rn marker and supplies that register to the formatter.

```
String 0: Address = %04X #R3
String 1: Row %u #R0 Data %02X #R1
```

```
SHOSTR 0 4 8
SHOSTR 1 4 20
UDSCRN
```

- Up to four register substitutions are actually formatted by the current implementation.
- Register markers are uppercase #R followed by digits.
- Match the number and type of format conversions to the register markers. Useful unsigned forms include %u, %X, %x, and width/zero padding such as %04X.
- String coordinates are converted to 8-bit values by SHOSTR.

Buttons and Indicator LEDs

V Pars has three physical buttons, but only **Buttons 2 and 3** are available to test programs. **Button 1 is reserved for leaving a running test early** and should not be used with BTNWT or BTNPR.

The buttons use **active-high logic**:

- LOW means the button is not pressed.
- HIGH means the button is pressed.

BTNWT — Wait for a Button

BTNWT button

BTNWT pauses the test until the selected button is pressed. Only Buttons 2 and 3 should be used.

BTNWT 2

This instruction waits while Button 2 is not pressed. When Button 2 becomes HIGH, execution continues with the next instruction.

BTNWT 3

This waits for Button 3.

An invalid or reserved button number should be ignored so the test continues without becoming stuck.

BTNPR — Jump When a Button Is Pressed

BTNPR button instructionRow

BTNPR checks the selected button once:

- If the button is pressed, execution jumps to the specified instruction row.
- If the button is not pressed, execution continues to the next instruction.

Only Buttons 2 and 3 should be used.

BTNPR 2 25

If Button 2 is pressed, the test jumps to compiled instruction row 25. Otherwise, the next instruction runs normally.

The target is a **one-based compiled instruction row**, matching the numbering shown beside executable instructions in VetusGUI. Blank lines, comments, and DIM declarations do not count as instruction rows.

LEDON — Turn On the Working LED

LEDON

LEDON turns on the tester's working indicator LED. Its operation follows the tester's saved LED setting.

LEDOFF — Turn Off the Working LED

LEDOFF

LEDOFF turns off the working indicator LED.

BLINKON — Start Blinking the Working LED

BLINKON speedMs

BLINKON starts blinking the working LED. The argument specifies the blink timing in milliseconds.

```
BLINKON 250
```

This starts the working LED blinking with a 250-millisecond timing value.

The timing value may be a literal, a DIM constant, or a register value:

```
BLINKON #R2
```

BLINKOFF — Stop Blinking the Working LED

```
BLINKOFF
```

BLINKOFF stops the blinking operation. Use it before ending the test when the LED no longer needs to blink.

Example

```
LEDON
```

```
BTNWT 2
```

```
LEDOFF
```

```
BLINKON 250
```

```
BTNPR 3 12
```

```
BLINKOFF
```

This sequence turns on the working LED, waits for Button 2, turns the LED off, starts it blinking, and checks whether Button 3 is pressed. If Button 3 is pressed, execution jumps to instruction row 12.

9. Add-on Board Commands and Safety

Vetus communicates with add-on boards through UART. A normal add-on sequence is:

```
BRD_ON
BRD_SEND_CMD command data1 data2
BRD_GET_RSLT resultSlot destinationRegister
BRD_OFF
```

BRD_ON must be used before sending commands or requesting results. When communication is finished, use BRD_OFF to close the UART connection.

Standard Commands Required by Add-on Boards

Every identifiable Vetus add-on board must support these standard commands:

Command	Purpose
0x01	Return the add-on board's identification number
0x02	Return the add-on board's name
0x03	Return the manufacturer's name

Board-specific commands must use other command numbers and must not replace or reuse Commands 1, 2, or 3.

Command 1 — Get the Board ID

Command 1 is required on every Vetus add-on board. Vetus uses it to determine whether an add-on board is connected and to identify which board it is.

A command that returns a number contains both the command number and the result slot:

0xCCSS

In this format:

- CC is the command number.
- SS is the result slot where the add-on board will store its answer.
- Result slots may be numbered from 0 through 63.

For Command 1, using result slot 1 produces:

0x0101

This means:

Command 0x01: Get the board ID

Result slot 0x01: Store the returned ID in slot 1

A complete test-code example is:

```
BRD_ON
BRD_SEND_CMD 0x0101 0 0
DELS 5
BRD_GET_RSLT 1 6
BRD_OFF
```

The sequence works as follows:

1. BRD_ON starts the add-on UART connection.
2. BRD_SEND_CMD 0x0101 0 0 sends Command 1 and instructs the add-on board to place its ID in result slot 1.

VETUS PARS | DEFINITIVE TEST-LANGUAGE GUIDE

3. DELMS 5 gives the add-on firmware time to process the command and store the result.
4. BRD_GET_RSLT 1 6 retrieves result slot 1 and stores its value in register R6.
5. BRD_OFF closes the UART connection.

The returned value can then be tested:

```
IF EQ #R6 0
    PASSFAIL 0
ENDIF
```

An ID of zero means that a valid add-on board was not identified. Each supported add-on board should return its assigned nonzero identification number.

Creating Additional Board Commands

Additional commands can use the same command-and-result-slot system, provided the firmware on the add-on board is programmed to recognize and translate them.

For example, suppose an add-on board defines Command 0x10 as “measure voltage” and stores the result in slot 2. The packed command would be:

0x1002

The Vetus test could use:

```
BRD_ON
BRD_SEND_CMD 0x1002 0 0
DELMS 5
BRD_GET_RSLT 2 4
BRD_OFF
```

The returned measurement would be stored in R4.

The add-on firmware must be programmed to:

1. Recognize Command 0x10.
2. Interpret data1 and data2 according to that command’s definition.
3. Perform the requested operation.
4. Store the resulting 32-bit value in result slot 2.
5. Return that slot when Vetus requests it with BRD_GET_RSLT.

Commands that do not return a value may be sent as a single command byte:

```
BRD_SEND_CMD 0x10 data1 data2
```

Commands that return a value should include the result slot:

```
BRD_SEND_CMD 0x1002 data1 data2
```

The meanings of data1 and data2 are determined by the add-on board’s firmware. They may contain settings, channel numbers, target values, timing values, bitfields, or other command-specific information.

All three BRD_SEND_CMD arguments may be literals, DIM constants, or register references.

Add-on Instruction Reference

Instruction	Operation
BRD_ON	Starts the UART connection used to communicate with the add-on board.
BRD_SEND_CMD <code>command data1 data2</code>	Sends a command and two 32-bit data values. The add-on firmware determines what the command and data values mean.
BRD_GET_RSLT <code>slot dest</code>	Requests a 32-bit value from result slot 0–63 and stores it in destination register 0–31. The destination is written as a bare register number, not <code>#Rn</code> .
BRD_OFF	Shuts down the add-on UART connection.

Add-on Board Safety

BRD_ON and BRD_OFF currently control the UART connection. They do not guarantee that electrical power has been removed from the add-on board.

Each add-on board's documentation must define:

- Its supported board-specific commands.
- The meaning of `data1` and `data2` for each command.
- Whether a command returns a result.
- Which result slot is used.
- How long the operation may take.
- Its voltage and current limits.
- Any required safe pin states.
- Any commands needed before disconnecting or ending a test.

The STM32 legacy reroute pin is intentionally kept in high-impedance mode so it does not interfere with older add-on-board wiring.

Early Vetus Pars Hardware Note

Early Vetus Pars units require a bodge wire to provide power to an add-on board. This hardware modification is not required on later revisions.

Before connecting an add-on board to an early unit, see the **Vetus Pars Add-on Board Quick Guide** for the correct wire location, routing, and safety instructions.

Safety architecture for test code

- Begin from ASAFE. Preload desired levels, then enable outputs.
- Never drive against a chip output. Configure bus direction before asserting enables, clocks, read/write strobes, or chip select.
- Use current limiting and board-specific voltage limits outside the language; no opcode can make an electrically unsafe fixture safe.
- Stop clocks before changing their pin purpose. Return strobes inactive, call BRD_OFF if used, then ASAFE.
- A firmware engine error ends the test and clears the running program path, but code should still be written with an explicit safe normal exit.
- Use PASSFAIL 1 only after every required check has succeeded. Default or error outcomes should remain fail.

10. Complete worked examples

Example A — simple input/output check

```
DIM DRIVE_PIN 3
DIM SENSE_PIN 4

ASAFE
SETPIN DRIVE_PIN 0
SETOUT DRIVE_PIN
SETIN SENSE_PIN

SETPIN DRIVE_PIN 1
DELUS 10
RDPIN SENSE_PIN 0

IF EQ #R0 1
    ASAFE
    PASSFAIL 1
    ENDTTEST
ENDIF

ASAFE
PASSFAIL 0
ENDTTEST
```

RDPIN's second argument is destination register slot 0. The IF reads it as #R0. Both exits return the socket to a safe state.

Example B — sweep an eight-bit address bus

```
DEF8 0 0x08070605 0x0C0B0A09
REGL0D 0 0

LOOP 256
    SET8 0 #R0 1
    DELUS 2
    REGINC 0
ENDLOOP

SET8 0 0 0
ASAFE
PASSFAIL 1
ENDTTEST
```

The mapping is pin 5=A0 through pin 12=A7. R0 naturally advances 0–255. SET8 uses only its low eight bits.

Example C — 16-bit address display

```
DEFADDR 0 0x08070605 0x0C0B0A09
DEFADDR 8 0x14131211 0x18171615
SETADDR 16 0x1234 1
REGL0D 3 0x1234
TXTCOL 0xFFFF 0x0000
SHOSTR 0 2 8
UDSCRN
DELMS 500
ASAFE
PASSFAIL 1
ENDTEST
```

```
// String 0: Address = %04X #R3
```

CONTINUED BLOW

11. Detailed opcode reference

Syntax columns show argument roles. “Value” accepts a literal, DIM, or #Rn unless the caution says otherwise. “Dest” is a bare register slot. Hex numbers in the Op column are the current compiled opcode byte.

SETPIN (0x01)

Syntax: SETPIN pin, level

Sets the output latch level of a user-numbered chip pin. It does not change direction.

Notes: Set/preload a level before SETOUT. Level should be 0 or 1.

```
SETPIN 5 1
```

SETIN (0x02)

Syntax: SETIN pin

Makes a chip pin an input through the mapped socket controls.

Notes: Use before RDPIN when the tester must not drive the pin.

```
SETIN 6
```

RDPIN (0x03)

Syntax: RDPIN pin, dest

Reads a chip pin and writes 0 or 1 to a destination register.

Notes: Destination is bare 0–31, not #Rn.

```
RDPIN 6 2
```

MULTIO (0x05)

Syntax: MULTIO pin1, pin2, pin3

Makes exactly three pins outputs.

Notes: Preload levels first. All three arguments are required.

```
MULTIO 2 3 4
```

MULTII (0x06)

Syntax: MULTII pin1, pin2, pin3

Makes exactly three pins inputs.

Notes: All three arguments are required.

```
MULTII 5 6 7
```

MULTIHI (0x07)

Syntax: MULTIHI pin1, pin2, pin3

Sets three pin latches HIGH.

Notes: Does not change direction.

```
MULTIHI 2 3 4
```

MULTILO (0x08)

Syntax: MULTILO pin1, pin2, pin3

Sets three pin latches LOW.

Notes: Does not change direction.

SETOUT (0x09)**Syntax:** SETOUT pin

Makes one chip pin an output.

Notes: Set the desired latch level first.

SETOUT 5

ASAFE (0x0B)**Syntax:** ASAFE

Places all 40 socket lines in Hi-Z and clears direction-shift state.

Notes: Use at startup and shutdown.

ASAFE

GOSTRT (0x0D)**Syntax:** GOSTRT

Sets the program counter to the first instruction.

Notes: This is a jump, not a complete engine-state reset; may loop forever.

GOSTRT

ALLDIR (0x0E)**Syntax:** ALLDIR sideA, sideB

Sets all package-pin directions from two bitfields. 1=output, 0=input.

Notes: Side A bit0=pin1; Side B bit0=pin N.

ALLDIR 0b0011 0b1000

ALLEVEL (0x0F)**Syntax:** ALLEVEL sideA, sideB

Sets all package-pin levels from two bitfields. 1=HIGH, 0=LOW.

Notes: Only low N/2 bits used; usually execute before ALLDIR.

ALLEVEL 0x03 0x08

LOOP (0x10)**Syntax:** LOOP count

Starts a counted loop.

Notes: Count must be at least 1. Maximum active records 32.

LOOP 100

ENDLOOP (0x11)**Syntax:** ENDLOOP

Decrements the nearest active counted loop and repeats or exits.

Notes: Must be paired with LOOP.

ENDLOOP

BRKLOOP (0x13)

Syntax: BRKLOOP

Exits the nearest enclosing LOOP, WHILE, or DOUNTIL.

Notes: Use structured nesting; avoid jumping into blocks.

```
BRKLOOP
```

DELUS (0x16)

Syntax: DELUS microseconds

Nonblocking engine delay in microseconds.

Notes: Very long values interact with timer wrap.

```
DELUS 2
```

DELMS (0x17)

Syntax: DELMS milliseconds

Nonblocking engine delay in milliseconds.

Notes: Converted to microseconds in 32-bit arithmetic.

```
DELMS 10
```

WHILE (0x18)

Syntax: WHILE condition, left, right

Tests before entry; skips to matching ENDWHL when false.

Notes: Unsigned comparison. DIV2 ignores right.

```
WHILE LT #R0 16
```

ENDWHL (0x19)

Syntax: ENDWHL

Returns to the matching WHILE for another test.

Notes: Must be paired.

```
ENDWHL
```

JMP (0x1A)

Syntax: JMP instructionRow

Jumps to a one-based compiled instruction row and rebuilds loop state.

Notes: Not a source line. Invalid/zero targets are ignored.

```
JMP 12
```

DOUNTIL (0x1B)

Syntax: DOUNTIL

Begins a loop that always executes once.

Notes: Maximum active do-until records 32.

```
DOUNTIL
```

ENDUNTIL (0x1C)

Syntax: ENDUNTIL condition, left, right

Repeats DOUNTIL body until the condition is true.

Notes: Unsigned comparison.

```
ENDUNTIL EQ #R2 1
```

CLKST (0x1D)

Syntax: CLKST pin, kHz

Starts the test clock on a pin.

Notes: One clock; GUI exposes kHz; maximum path clamps at 500 kHz.

```
CLKST 8 100
```

CLKSTP (0x1E)

Syntax: CLKSTP

Stops the active test clock.

Notes: Stop before repurposing the pin.

```
CLKSTP
```

SETADDR (0x21)

Syntax: SETADDR nbits, value, direction

Drives or inputs the defined A0–A31 map.

Notes: Use continuous mapping; direction-only update caveat in Chapter 5.

```
SETADDR 16 0x1234 1
```

DEFADDR (0x22)

Syntax: DEFADDR base, packedLo, packedHi

Defines eight consecutive address-bit pin mappings.

Notes: Base 0,8,16,24; low byte is first logical bit.

```
DEFADDR 0 0x08070605 0x0C0B0A09
```

ARRADD (0x25)

Syntax: ARRADD array, index, value

Writes/appends an item to 32-word array A or B.

Notes: array 0=A; nonzero=B. Index 0 has append semantics after first item.

```
ARRADD 0 0 25
```

ARRGET (0x26)

Syntax: ARRGET array, index, dest

Copies an existing array item to a register.

Notes: Out-of-length read leaves destination unchanged.

```
ARRGET 0 1 3
```

GETADDR (0x27)

Syntax: GETADDR nbits, dest

Reads the continuous A0-up address map into a register.

Notes: Current reader ignores nbits and stops at first unmapped bit.

```
GETADDR 16 3
```

PHIZ (0x28)**Syntax:** PHIZ pin

Requests Hi-Z on one lower-level index.

PHIZ 5

TOGPIN (0x29)**Syntax:** TOGPIN pin, mode

Pulses low-high-low when mode=1; high-low-high when mode=0, with 1 ms steps.

TOGPIN 4 1

PIN2ARR (0x2A)**Syntax:** PIN2ARR array, packedLo, packedHi

Appends eight packed pin-byte values to A or B.

Notes: Zero bytes are appended too.

PIN2ARR 0 0x04030201 0x08070605

DEF8 (0x2B)**Syntax:** DEF8 map, packedLo, packedHi

Defines one eight-bit map.

Notes: Use map 0–5 only despite broader GUI validation.

DEF8 0 0x08070605 0x0C0B0A09

SET8 (0x2C)**Syntax:** SET8 map, value, direction

Sets direction and optionally drives an eight-bit map.

Notes: Value is truncated to 8 bits; map 0–5.

SET8 0 0xA5 1

GET8 (0x2D)**Syntax:** GET8 map, dest

Reads an eight-bit map to a register.

Notes: Use valid map 0–5 and dest 0–31; bounds are incomplete.

GET8 0 4

REGL0D (0x40)**Syntax:** REGL0D dest, value

Loads a value into a register.

REGL0D 3 0x20

REGINC (0x42)**Syntax:** REGINC dest

Adds 1 to a register with 32-bit wrap.

Notes: 0xFFFFFFFF becomes 0.

REGINC 3

REGDEC (0x43)

Syntax: REGDEC dest

Subtracts 1 with 32-bit wrap.

Notes: 0 becomes 0xFFFFFFFF.

```
REGDEC 3
```

ADD (0x44)

Syntax: ADD left, right, dest

Unsigned 32-bit addition.

Notes: Wraps on overflow; dest should be 0–31.

```
ADD #R1 5 2
```

SUBTR (0x45)

Syntax: SUBTR left, right, dest

Unsigned subtraction, clamped at zero when right > left.

Notes: It does not wrap negative results.

```
SUBTR #R1 5 2
```

MULT (0x46)

Syntax: MULT left, right, dest

Unsigned 32-bit multiplication.

Notes: Wraps on overflow.

```
MULT #R1 8 2
```

DIV (0x47)

Syntax: DIV left, right, dest

Unsigned integer division.

Notes: Avoid zero divisor; helper error is not fully promoted to engine failure.

```
DIV #R1 8 2
```

RNG (0x49)

Syntax: RNG min, max, dest

Generates an inclusive pseudo-random value.

Notes: min≤max; full 0..0xFFFFFFFF range overflows range calculation.

```
RNG 1 6 2
```

AND (0x4A)

Syntax: AND left, right, dest

Bitwise AND.

```
AND #R1 0xFF 2
```

OR (0x4B)

Syntax: OR left, right, dest

Bitwise OR.

OR #R1 0x80 2

SHL (0x4C)

Syntax: SHL value, count, dest

Logical left shift.

Notes: Keep count 0–31; larger C shifts are undefined.

SHL 1 7 2

SHR (0x4D)

Syntax: SHR value, count, dest

Logical right shift.

Notes: Keep count 0–31.

SHR #R1 4 2

NOT (0x4E)

Syntax: NOT value, dest

Bitwise complement.

Notes: All 32 bits are inverted.

NOT #R1 2

XOR (0x4F)

Syntax: XOR left, right, dest

bitwise XOR.

XOR #R1 #R2 3

IF (0x50)

Syntax: IF condition, left, right

Executes body only when comparison is true.

Notes: Pair with ENDIF; no ELSE support.

IF EQ #R0 1

ENDIF (0x51)

Syntax: ENDIF

Ends an IF block.

ENDIF

SHOSTR (0x60)

Syntax: SHOSTR string, x, y

Draws one of eight strings at coordinates, with register formatting.

Notes: Up to four effective register substitutions. Follow with UDSCRN.

SHOSTR 0 4 8

FILSCR (0x61)

Syntax: FILSCR RGB565

Fills the display buffer with a color.

Notes: Follow with UDSCRN to show it.

```
FILSCR 0x0000
```

TXTCOL (0x63)

Syntax: TXTCOL foreground, background

Sets RGB565 text colors.

```
TXTCOL 0xFFFF 0
```

PROG (0x64)

Syntax: PROG percent, fill, background

Draws the standard progress bar.

Notes: Percent clamps to 100; colors are RGB565.

```
PROG 50 0x07E0 0
```

BLINKON (0x68)

Syntax: BLINKON speedMs

Starts working-LED blinking.

Notes: Speed is resolved value.

```
BLINKON 250
```

BLINKOFF (0x69)

Syntax: BLINKOFF

Stops working-LED blinking.

```
BLINKOFF
```

BTNPR (0x6A)

Syntax: BTNPR button, target

Conditionally changes program counter based on button state.

```
BTNPR 1 20
```

BTNWT (0x6B)

Syntax: BTNWT button

Stalls based on button 1–3 GPIO state.

```
BTNWT 1
```

LEDON (0x6C)

Syntax: LEDON

Requests working LED on.

Notes: Behavior is conditioned by saved LED setting.

```
LEDON
```

LEDOFF (0x6D)

Syntax: LEDOFF

Turns working LED output off.

```
LEDOFF
```

UDSCRN (0x6F)**Syntax:** UDSCRN

Waits for display availability and requests DMA buffer update. A short delay should come after (~10ms)

UDSCRN

SEVSEG (0x70)**Syntax:** SEVSEG value, packedXY, packedSizeColor

Draws a pseudo seven-segment number.

Notes: Pack fields exactly as described in Chapter 8.

SEVSEG #R0 0x00140028 0x0002F800

SHOWIMG (0x71)**Syntax:** SHOWIMG image, x, y

Loads and draws a stored zero-based image.

Notes: Start must be on-screen; follow with UDSCRN and DELMS 10.

SHOWIMG 11 40 4

BRD_ON (0xA0)**Syntax:** BRD_ON

Initializes current add-on UART transport.

BRD_ON

BRD_OFF (0xA1)**Syntax:** BRD_OFF

Shuts down current add-on UART transport.

BRD_OFF

BRD_SEND_CMD (0xA3)**Syntax:** BRD_SEND_CMD command, data1, data2

Sends a four-word packet; fourth word is zero.

Notes: Timeout fails the test. Meanings are board-specific.

BRD_SEND_CMD 0x0101 0 0

BRD_GET_RSLT (0xA4)**Syntax:** BRD_GET_RSLT slot, dest

Requests board result slot 0–63 into register 0–31.

Notes: Timeout or invalid slot/dest fails.

BRD_GET_RSLT 1 4

FUNC (0xC1)

Syntax: FUNC name

Marks a named function entry; GUI packs its name and resolves CALLs.

Notes: Put functions after ENDTEST. Marker is a runtime NOP.

```
FUNC pulse
```

ENDFUNC (0xC2)

Syntax: ENDFUNC

Returns from the active function.

Notes: No effect if not in a CALL; nested calls unsupported.

```
ENDFUNC
```

CALL (0xC3)

Syntax: CALL name

Calls a GUI-resolved function row.

Notes: Function must exist; nesting/recursion unsupported.

```
CALL pulse
```

ENDTEST (0xF0)

Syntax: ENDTEST

Stops test, clears program execution, and opens result screen.

Notes: Set PASSFAIL first.

```
ENDTEST
```

PASSFAIL (0xF2)

Syntax: PASSFAIL value

Sets result flag: zero=fail, nonzero=pass.

Notes: behavior is pass when nonzero.

```
PASSFAIL 1
```