

Vetus GUI Integration Guide

Host communication, packet mechanics, and command order

Purpose and compatibility

This is the host-side contract needed to build a functional replacement for VetusGUI. It documents every operation currently exposed by the GUI: connecting, identifying and refreshing the device, editing and saving local content, compiling and sending test code, saving one chip or a batch, loading and deleting chips, managing images and fonts, and reading or changing settings.

The guide describes public wire behavior and file interchange formats. It does not reproduce the GUI source, firmware implementation, flash addresses, or test-engine internals. Keep command constants and payload layouts in one protocol module so a firmware-specific profile can be selected if a future release changes a structure.

Host architecture

A robust GUI should have five independent layers:

1. A serial transport that opens and closes the USB CDC port and performs complete binary writes.
2. A frame codec that applies CRC-8, COBS, and the zero delimiter.
3. A transaction controller that owns the sequence counter, timeouts, retries, and matching responses.
4. Workflow state machines for chips, test code, images, fonts, and settings.
5. UI models and local file codecs. UI controls should never write raw serial bytes directly.

Permit only one ordered transfer step at a time. Ordinary list requests can also be serialized; this makes reconnects and error reporting predictable.

Connection and refresh

Vetus appears as a USB CDC serial device.

- Baud rate: **115200**
- Data bits: **8**
- Parity: **none**
- Stop bits: **1**
- Flow control: **none**
- Mode: raw binary; do not add line endings or translate bytes

On connect, clear any incomplete receive frame and reset transaction state. A practical refresh sequence is:

1. Send `0xDC` to read the serial number.
2. Send `0xDB` to read the firmware version.
3. Send `0xC8`, then `0xC7` once per returned chip index.
4. Send `0xDE` for font slots 0 through 5.
5. Send `0xD0`, then scan image names with `0xD4` for the slots the UI presents.
6. Send `0xE4` with operation `0x00` to read settings.

Refresh after any write or delete that can change indexes. If the port disappears, cancel the active operation, close the handle, refresh the port list, and reconnect only to an unambiguous matching device. The reference GUI makes at most three automatic reconnect attempts.

Packet framing

Decoded packet

Byte(s)	Meaning
0	Command or response code
1	Unsigned 8-bit sequence number
2 through N	Optional payload
Last byte	CRC-8 over every preceding decoded byte

Wire packet

1. Build [code, sequence, payload...].
2. Calculate CRC-8 over that array and append it.
3. COBS-encode the entire decoded packet including the CRC.
4. Append one `0x00` frame delimiter.
5. Write all bytes. A partial operating-system write is not a complete packet.

There is no outer length field. Serial reads may split a frame or combine several frames, so maintain a persistent receive buffer and split only on `0x00`.

Exact CRC-8 C code

Parameters are polynomial `0x07`, initial value `0x00`, no reflection, and no final XOR.

```
uint8_t computeCRC8(const uint8_t *data, uint32_t length)
{
    uint32_t crc = 0x00;

    for (uint32_t i = 0; i < length; i++)
    {
        crc ^= data[i];

        for (int j = 0; j < 8; j++)
        {
            if (crc & 0x80)
                crc = ((crc << 1) ^ 0x07) & 0xFF;
            else
                crc = (crc << 1) & 0xFF;
        }
    }

    return (uint8_t)crc;
}
```

Exact COBS C code

The caller supplies a sufficiently large output buffer. Decode returns `-1` for an invalid frame; otherwise both routines return the number of output bytes.

```

int COBS_Decode(const uint8_t *input, int length, uint8_t *output) {
    int read_index = 0;
    int write_index = 0;

    while (read_index < length) {
        uint8_t code = input[read_index++];
        if (code == 0 || read_index + code - 1 > length) {
            return -1; // invalid
        }
        for (int i = 1; i < code; i++) {
            output[write_index++] = input[read_index++];
        }
        if (code != 0xFF && read_index < length) {
            output[write_index++] = 0x00;
        }
    }
    return write_index;
}

int COBS_Encode(const uint8_t *input, int length, uint8_t *output) {
    const uint8_t *end = input + length;
    uint8_t *code_ptr = output++;
    uint8_t code = 1;

    uint8_t *output_start = code_ptr;

    while (input < end) {
        if (*input == 0) {
            *code_ptr = code;
            code_ptr = output++;
            code = 1;
            input++;
        } else {
            *output++ = *input++;
            code++;
            if (code == 0xFF) {
                *code_ptr = code;
                code_ptr = output++;
                code = 1;
            }
        }
    }
    *code_ptr = code;
    return output - output_start;
}

```

Language-neutral send and receive mechanics

```

sendCommand(code, payload):
    seq = nextSequence
    nextSequence = (nextSequence + 1) modulo 256
    body = [code, seq] + payload
    decoded = body + [CRC8(body)]
    SERIAL_WRITE_ALL(COBS_ENCODE(decoded) + [0x00])
    return seq

onSerialBytes(bytes):
    append bytes to persistent encodedBuffer
    while encodedBuffer contains 0x00:
        remove bytes through first delimiter as one encoded frame
        ignore an empty frame
        decoded = COBS_DECODE(frame without delimiter)
        reject if invalid or shorter than 3 bytes
        body = decoded except final byte
        reject unless CRC8(body) equals final byte
        dispatch code=body[0], sequence=body[1], payload=body[2...]

```

Use a receive-buffer ceiling, such as 4096 encoded bytes. If no delimiter arrives before the ceiling, clear the partial frame and report a framing error.

Transaction and error rules

- Increment the sequence for every newly issued packet; wrapping 255 to 0 is valid.
- Match the reply by sequence number as well as the expected reply type.
- During a multi-step upload, wait for each reply before sending the next packet.
- Use a response timeout of roughly 3 to 5 seconds. A timeout aborts the current workflow.
- Ignore late packets that do not match the active transaction.
- If `0xB1` requests retry, resend the identical command and payload with a new sequence number.
- Never interpret binary packets as console text or send console text through this channel.

Response	Meaning	Required host action
<code>0xB0</code>	ACK/received	Advance only when that workflow accepts ACK
<code>0xB1</code>	Retry	Resend the current step exactly, with a new sequence
<code>0xB2</code>	Done	Advance to the next step
<code>0xB3</code>	Failed	Abort and show the operation that failed
<code>0xB4</code>	Bad payload size	Abort; correct the client payload
<code>0xB5</code>	Still erasing	Keep the erase operation active and wait or poll

Read commands normally return a typed data response rather than `0xB2`.

Command directory

All multi-byte integers are unsigned little-endian unless stated otherwise. Indexes are zero-based.

Identification, mode, and settings

Operation	Send	Payload	Response
Firmware version	<code>0xDB</code>	none	<code>0xBF</code> , four bytes [build, patch, minor, major]
Serial number	<code>0xDC</code>	none	<code>0xE0</code> , UTF-8/ASCII bytes; no NUL is required
Check upload mode	<code>0xAF</code>	none	<code>0x05</code> ready or <code>0x06</code> not ready
Read settings	<code>0xE4</code>	[<code>0x00</code>]	<code>0xE5</code> , 14-byte snapshot
Set a setting	<code>0xE4</code>	[<code>0x01</code> , settingID, value]	<code>0xE5</code> , complete updated snapshot

Chip and test-code commands used by the GUI

Operation	Send	Payload	Response
Chip count	0xC8	none	0xB9 [countLo,countHi]
Chip name	0xC7	[indexLo,indexHi]	0xBA, NUL-terminated UTF-8 name
Begin test metadata	0xCA	41-byte test metadata	0xB2
Begin test strings	0xAC	none	0xB2
Test strings chunk	0xAD	up to 63 bytes	0xB2
Finish test strings	0xD9	none	0xB2
Declare test-code length	0xCD	[lengthLo,lengthHi]	0xB2
Test-code chunk	0xCB	up to 63 bytes	0xB2
Run prepared RAM test	0xCC	none	0xB2 or failure
Save prepared test as chip	0xC6	none	0xB2
Finish batch save	0xCF	none	0xB2
Load chip metadata	0xD6	index, then [0xAA,0xAA] continuations	0xBC
Load chip code	0xD7	index, then [0xAA,0xAA] continuations	0xBD
Load one chip string	0xD8	[chipLo,chipHi,stringIndex]	0xBE, 64 bytes
Delete chip	0xDA	[indexLo,indexHi]	0xB2
Delete all chips	0xD3	none	status response

The firmware also recognizes older list aliases 0xD2 for count and 0xD5 for name, plus the older staged chip-upload family 0xA6 through 0xAB. New GUI implementations should use the commands in the table because they match the current VetusGUI workflow.

Image commands

Operation	Send	Payload	Response
Image count	0xD0	none	0xB7 [countLo,countHi]
Image name	0xD4	[slotLo,slotHi]	0xBB name; 0xB3 if empty/invalid
Image metadata	0xE7	[slotLo,slotHi]	0xE9, 38 bytes
Image data chunk	0xE8	[slotLo,slotHi,offsetLo,offsetHi]	0xEA [offsetLo,offsetHi,data...]
Begin metadata upload	0xA0	[0x80,0x00] (128 bytes)	0xB2/ACK
Select destination slot	0xEC	[slotLo,slotHi]	0xB2

Operation	Send	Payload	Response
Metadata chunk	0xA1	up to 48 bytes	0xB2
Commit metadata	0xA2	none	0xB2
Begin image data	0xA3	[lengthLo, lengthHi]	0xB2
Image data chunk	0xA4	up to 48 bytes	0xB2
Commit image data	0xA5	none	0xB2
Delete image	0xEB	[slotLo, slotHi]	0xB2
Erase user images	0xD1	none	status response

Font commands

Operation	Send	Payload	Response
Font-slot metadata	0xDE	[slot], 0 through 5	0xE2 metadata or 0xE1 [slot] empty
Begin metadata upload	0xC0	[0x80, 0x00, slot]	0xB2/ACK
Metadata chunk	0xC1	up to 48 bytes	0xB2
Commit metadata	0xC2	none	0xB2
Begin font data	0xC3	[lengthLo, lengthHi]	0xB2
Font data chunk	0xC4	up to 48 bytes	0xB2
Commit font data	0xC5	none	0xB2
Delete font	0xE6	[slot]	0xB2 or failure

0xDF is reserved as a font-data read command in the protocol header, but the current GUI does not use font-data download. Do not present device font editing as supported unless the target firmware implements and verifies that response path.

Practical command by command behavior

In this section, “payload” means only the bytes after command and sequence and before CRC. The framing layer still adds command, sequence, CRC, COBS encoding, and the final zero. For example, `SEND CA + 41 metadata bytes` means the decoded body begins `[CA, sequence, ...41 bytes...]`.

Device identity and readiness

DB get firmware version

Send: command DB, no payload.

PCB action: packs the four firmware version components into one 32-bit value.

Receive: BF with exactly four payload bytes in this order: build, patch, minor, major. Display them as `major.minor.patch.build`.

Next: select the matching protocol profile, then continue the connection refresh. Treat any other response or payload length as an identification failure.

DC get serial number

Send: command DC, no payload.

PCB action: reads and returns its serial text.

Receive: E0 followed by the serial characters. Do not require a terminating zero.

Next: store the serial as the device identity. It can be used to recognize the same unit after a reconnect.

AF check upload mode

Send: command AF, no payload.

PCB action: checks whether the Vetus screen is currently in the mode that accepts uploaded tests and assets.

Receive: 05 means continue; 06 means do not send upload data.

Next: after 05, start the first command of the selected upload workflow. After 06, stop and tell the user to open Upload Mode on Vetus.

Chip list, load, and deletion

C8 get chip count

Send: command C8, no payload.

PCB action: reads the stored-chip count.

Receive: B9 with [count low, count high].

Next: if count is nonzero, request name zero with C7; continue one name at a time through count minus one. D2 is the older count alias and is not needed by a new GUI.

C7 get chip name

Send: command C7, payload [index low, index high].

PCB action: loads that chip's metadata and returns its name.

Receive: BA with name bytes, normally ending at the first 00.

Next: store the name at that exact index. Send C7 for the next index, or finish the list refresh. D5 is the older name alias.

D6 get chip metadata

First send: command D6, payload [index low, index high].

PCB action: selects the chip and prepares its 128-byte metadata block.

First receive: BC with no metadata bytes. This is only the selection acknowledgement.

Continuation send: command D6, payload AA AA.

PCB action: returns the next portion of the selected metadata.

Continuation receive: BC with up to 57 metadata bytes.

Next: repeat D6 AA AA until exactly 128 bytes have been collected. Do not look for a special last-packet marker. Validate bytes 0 and 1 as 5A A5, then read the real code size from offsets 38 and 39. After valid metadata, begin code loading with D7 and the original chip index.

D7 get chip code

First send: command D7, payload [index low, index high].

PCB action: selects the chip code and calculates its 16-byte-aligned transfer length.

First receive: BD with [padded length low, padded length high] and no code.

Continuation send: command D7, payload AA AA.

Continuation receive: BD with up to 48 code bytes.

Next: repeat until the advertised padded length is collected. Trim the buffer to the real code length found in metadata. Then request strings 0 through 7 with D8.

D8 get one chip string

Send: command D8, payload [chip index low,chip index high,string index]. String index must be 0 through 7.

PCB action: reads one fixed 64-byte string slot.

Receive: BE with exactly 64 bytes.

Next: decode through the first 00, then request the next string. After string 7, the chip load is complete.

DA delete one chip

Send: command DA, payload [index low,index high].

PCB action: deletes the selected record. If it was not the last record, the PCB moves the last chip into the empty index.

Receive: B2 on success or a failure status.

Next: discard the entire cached chip list and request C8 again. Do not merely remove one row locally because the moved record changes an index.

D3 delete all chips

Send: command D3, no payload, only after explicit user confirmation.

PCB action: erases the chip area and resets the stored count.

Receive: completion or erase-status responses. B5 means the erase is still underway, not that it failed.

Next: keep the UI locked until completion, then request C8 again.

Preparing, running, and saving a test

CA send test metadata

Send: command CA with exactly 41 payload bytes:

00-01	5A A5	Valid marker, little-endian A55A
02-33	name[32]	UTF-8 bytes, zero-padded when shorter
34-35	type low,high	Unsigned 16-bit chip type
36	pin count	Unsigned byte
37	image index	Unsigned byte
38-39	code size low,high	
40	non-empty string count	

5A A5 is not an arbitrary wake-up code. It is the little-endian byte form of the valid-record tag A55A. It appears at metadata offsets 0 and 1 so the PCB recognizes the record as valid. Do not transmit A5 5A.

PCB action: resets the temporary upload state, copies the 41-byte record, and switches to the GUI test screen.

Receive: B2.

Next: send AC to clear and begin the 512-byte string area.

AC begin test strings

Send: command AC, no payload.

PCB action: clears its temporary 512-byte string buffer and resets the string offset to zero.

Receive: B2.

Next: send the first AD chunk.

AD send test string bytes

Send: command AD with 1 through 63 bytes. The combined transfer must be exactly 512 bytes: eight consecutive 64-byte slots.

PCB action: appends the payload at the current temporary string offset.

Receive: B2, B1 to repeat this exact chunk, or an error.

Next: after B2, advance the offset by the payload length. Send another AD until offset 512; then send D9. Never advance the offset after B1.

D9 finish test strings

Send: command D9, no payload.

PCB action: verifies that exactly 512 bytes arrived, splits them into eight 64-byte strings, and guarantees a terminator at the end of each slot.

Receive: B2 if the length is exact; otherwise a failure.

Next: send CD with the compiled code length.

CD declare test length

Send: command CD, payload [code length low,code length high].

PCB action: clears the RAM test-code buffer, stores the expected length, and resets the code offset.

Receive: B2.

Next: if length is nonzero, send the first CB code chunk.

CB send test code

Send: command CB with 1 through 63 compiled-code bytes.

PCB action: appends the bytes to the RAM test buffer.

Receive: B2, B1 retry, or an error.

Next: after B2, advance by the payload length and send the next chunk. When the declared number of bytes has been acknowledged, the test is prepared. Choose either CC to run it or C6 to save it.

CC run the prepared test

Send: command CC, no payload.

PCB action: loads the prepared RAM program into the test engine and begins the GUI-provided test. A zero-length program fails.

Receive: B2 when accepted or a failure response.

Next: show that the test was started. Running does not store a new chip.

C6 save the prepared test as a chip

Send: command C6, no payload, only after CA, all string commands, CD, and all CB chunks succeeded.

PCB action: commits the prepared metadata, strings, and actual received code to the next chip slot. It writes the valid tag, sets the real code size, stores eight string slots, and increments the chip count.

Receive: B2 on success.

Next: for a single save, request C8 and rebuild the list. For a batch, begin the next chip at CA.

CF finish batch

Send: command CF, no payload after the final chip's C6 succeeds.

PCB action: closes the batch-save operation and updates the device screen/state.

Receive: B2.

Next: request C8 and rebuild the complete list.

Images

D0 get image count

Send: command **D0**, no payload.

PCB action: reads the image count.

Receive: B7 [count low, count high].

Next: scan the GUI-visible slots with **D4**. Treat the count as information; slot occupancy is established by the per-slot name replies.

D4 get image name

Send: command **D4**, payload [slot low, slot high].

PCB action: checks that slot's metadata.

Receive: BB with a NUL-terminated name, or B3 for an empty or invalid slot.

Next: mark the slot occupied or empty, then request the next displayed slot.

E7 get image metadata

Send: command **E7**, payload [slot low, slot high].

PCB action: reads the selected image metadata.

Receive: E9 with exactly 38 bytes: name at 0..31, width at 32, height at 33, and compressed size at 34..37 little-endian.

Next: validate the values, set data offset to zero, and send **E8**.

E8 get image data

Send: command **E8**, payload [slot low, slot high, offset low, offset high].

PCB action: reads from that compressed-data offset.

Receive: EA containing [returned offset low, returned offset high] followed by up to 61 bytes.

Next: reject an unexpected returned offset. Otherwise append the data and add the received data-byte count to the offset. Repeat until the metadata size is reached, then decode RGB565 RLE.

A0 begin image metadata upload

Send: command **A0**, payload **80 00**. This is the little-endian value 128, the exact image metadata size.

PCB action: resets its image-upload state and prepares a 128-byte metadata buffer.

Receive: B2 or ACK.

Next: send **EC** to select the destination slot.

EC select image slot

Send: command **EC**, payload [slot low, slot high]. The current GUI allows 1 through 239; slot 0 is rejected.

PCB action: records where the completed image will be stored.

Receive: B2.

Next: send the first **A1** metadata chunk.

A1 send image metadata bytes

Send: command **A1**, up to 48 bytes. Send the complete 128-byte block in order.

PCB action: appends metadata at its current offset.

Receive: B2, B1, or failure.

Next: advance only after success. When 128 bytes are acknowledged, send **A2**.

A2 commit image metadata

Send: command **A2**, no payload.

PCB action: validates that all metadata bytes arrived and prepares the data stage.

Receive: **B2**.

Next: send **A3** with the RLE byte length.

A3 begin image data

Send: command **A3**, payload [RLE length low, RLE length high].

PCB action: clears the temporary image-data buffer and records the required length.

Receive: **B2**.

Next: send **A4** chunks.

A4 send image data bytes

Send: command **A4**, up to 48 RLE bytes.

PCB action: appends the bytes in order.

Receive: **B2**, **B1**, or failure.

Next: after the declared number is acknowledged, send **A5**.

A5 save image

Send: command **A5**, no payload.

PCB action: validates lengths, writes the image and metadata to the selected slot, and marks the metadata valid.

Receive: **B2**.

Next: refresh image names and optionally load the slot back with **E7** and **E8** to verify it.

EB delete image and **D1** erase user images

Send EB: **EB** [slot low, slot high]; only slots 15 through 239 may be individually deleted.

PCB action: invalidates that user image.

Receive: **B2** or failure. Refresh slot names next.

Send D1: command **D1**, no payload, after confirmation.

PCB action: erases user images while preserving protected slots 0 through 14.

Receive: erase status and completion. Keep waiting through **B5**, then refresh all image slots.

Fonts

DE get font slot

Send: command **DE**, payload [slot], where slot is 0 through 5.

PCB action: checks and reads that font's metadata.

Receive: **E1** [slot] if empty, or **E2** with slot, width, height, glyph count, four-byte data size, and 32-byte name as defined later in this guide.

Next: store the slot result and request the next slot. After slot 5, the font list is complete.

C0 begin font metadata upload

Send: command **C0**, payload **80 00 slot**. **80 00** declares exactly 128 metadata bytes.

PCB action: clears the font-upload state and records the destination slot.

Receive: **B2** or **ACK**.

Next: send **C1** metadata chunks.

C1 send font metadata bytes

Send: command C1, up to 48 bytes, until all 128 bytes have been sent.

PCB action: appends metadata at the current offset.

Receive: B2, B1, or failure.

Next: after byte 128 is acknowledged, send C2.

C2 commit font metadata

Send: command C2, no payload.

PCB action: validates the complete metadata block and prepares the data stage.

Receive: B2.

Next: send C3.

C3 begin font data

Send: command C3, payload [font byte length low, font byte length high].

PCB action: clears the font-data buffer and stores the expected length.

Receive: B2.

Next: send C4 chunks.

C4 send font data bytes

Send: command C4, up to 48 column bytes.

PCB action: appends the font bytes.

Receive: B2, B1, or failure.

Next: after the exact declared length is acknowledged, send C5.

C5 save font

Send: command C5, no payload.

PCB action: commits the font data and metadata to the chosen slot and marks it valid.

Receive: B2.

Next: send DE [slot] and compare the returned metadata with the upload.

E6 delete font

Send: command E6, payload [slot].

PCB action: rejects the request if this is the active/default font; otherwise it invalidates the slot.

Receive: B2 or failure.

Next: refresh slots 0 through 5. If the font is active, first set setting ID 07 to another populated slot, wait for E5, and retry deletion.

E4 settings command

Read all settings

Send: E4 00.

PCB action: reads its stored settings.

Receive: E5 with all 14 setting bytes.

Next: update every settings control from the returned bytes.

Set one setting

Send: E4 01 settingID value.

PCB action: validates and saves that field, then returns the complete settings record.

Receive: E5 with all 14 bytes, or failure.

Next: update the entire settings screen from E5; do not leave the requested value displayed merely because it was sent.

Settings model

0xE5 always returns a complete 14-byte settings snapshot. Update the GUI from this response instead of assuming the requested value was accepted.

Offset	Meaning
3	Startup-logo value
4	Logo speed
7	LED value
8	AUX/fan value
9	Default font slot

Setting IDs used with [0x01, id, value] are:

ID	Setting	Current GUI encoding/validation
0x02	Startup logo	GUI sends 0 for Show, 1 for Hide
0x03	Logo speed	0 slow, 1 medium, 2 fast
0x05	LEDs	GUI sends 0 for On, 1 for Off
0x06	AUX/fan	GUI sends 1 for On, 0 for Off
0x07	Default font	Slot 0 through 5; slot must exist and be populated

Settings are stored by the device. After setting a value, retain the matching 0xE5 snapshot as the authoritative state.

Chip records and local chip files

Device chip metadata

The stored chip metadata block is 128 bytes.

Offset	Size	Meaning
0	2	Valid tag, little-endian 0xA55A
2	32	UTF-8 name field; NUL-terminate when shorter than 32
34	2	Chip type, little-endian
36	1	Pin count
37	1	Image index

Offset	Size	Meaning
38	2	Real code length, little-endian
40	1	String count
41	87	Reserved; write zero

The test-upload command `0xCA` sends only bytes 0 through 40 of this layout, exactly 41 bytes. For a new prepared test, use tag bytes `[0x5A, 0xA5]`, a UTF-8 name of at most 32 bytes, and the actual compiled-code length. The save command normalizes the committed tag and lengths.

Local editable chip format

Local chip documents are UTF-8 text and end with a newline. Code lines, blank lines, leading spaces, and comments are preserved. Strings are restricted to one line when written; embedded newlines become spaces. Unknown metadata keys may be ignored for forward compatibility.

```
[chipfile]
type = single

[chips]
----

[meta]

name = Example Chip
pins = 14
type = 0
imageIndex = 1

[code]

...editable test source...

[strings]

0=First string
1=
2=
3=
4=
5=
6=
7=

[ENDCHIPS]
```

type in `[chipfile]` must be `single`. Missing pin/type/image values may fall back to application defaults, but a writer should always include them.

Compiling and sending test code

The GUI edits source rows, validates arguments, resolves functions, and emits a compact binary program. Each executable instruction is exactly 16 bytes: four unsigned 32-bit little-endian words.

Word	Meaning
0	Opcode

Word	Meaning
1	Argument 1 or zero
2	Argument 2 or zero
3	Argument 3 or zero

Raw instructions accept no more than three arguments. A register reference is encoded as `0x80000000` with the register value appended to the `0x80000000` OR registerNumber. A CALL target is the one-based source instruction row containing the matching FUNC. A FUNC name is limited to 12 bytes and is packed across the three argument words. The compiled program must fit both the 16-bit length field and the device's 4096-byte RAM test buffer.

The protocol transports compiled code, not source text. A replacement GUI may implement its own editor/compiler or import already compiled records, but it must emit the same 16-byte instruction records. Instruction meanings belong in the separate Vetus programming reference; the transport does not parse them.

String block

Always build exactly eight 64-byte string slots, for a total of 512 bytes. For each slot, write at most 63 UTF-8 bytes, append a NUL, and pad the remainder with zero. Empty strings are 64 zero bytes. The metadata string-count byte is the number of non-empty editor strings, although the device save path stores all eight slots.

Exact send sequence

1. Send `0xAF`. Continue only after matching `0x05`; on `0x06`, tell the user to open Upload Mode on Vetus.
2. Compile and validate the source before writing anything.
3. Send `0xCA` with the 41-byte test metadata; wait for `0xB2`.
4. Send `0xAC` with no payload; wait for `0xB2`.
5. Send all 512 string bytes using `0xAD` chunks of at most 63 bytes, waiting for `0xB2` after every chunk.
6. Send `0xD9` with no payload; wait for `0xB2`. The device rejects a string transfer that is not exactly 512 bytes.
7. Send `0xCD` with the compiled code length as two little-endian bytes; wait for `0xB2`.
8. Send code using `0xCB` chunks of at most 63 bytes, waiting for `0xB2` after every chunk.
9. The program is now prepared in RAM. It is not yet a saved chip.

Run versus save

- To execute the prepared RAM test from software, send `0xCC` and wait for `0xB2`. The current GUI can also direct the user to start the prepared test with the Vetus controls.
- To store the prepared content as the next chip, send `0xC6` and wait for `0xB2`, then refresh the entire chip list.
- Do not send `0xC6` before every metadata, string, and code step has completed.

Batch save

1. Check upload mode once.
2. For each local chip, perform the complete metadata, strings, and code sequence above.
3. Send `0xC6` after each prepared chip and wait for `0xB2`.
4. After the last chip, send `0xCF` and wait for `0xB2`.
5. Refresh the full chip list.

If any step fails or times out, stop the batch at that item and report its name and stage. Never continue from a guessed offset.

Reading and deleting chips

Refresh the chip list

1. Send `0xC8`; parse `0xB9` payload as a 16-bit count.
2. For index 0 through count minus 1, send `0xC7 [indexLo, indexHi]`.
3. Decode each matching `0xBA` payload through the first NUL.

The current capacity is 2975 chip records. Do not allocate based only on an untrusted response without applying a reasonable bound.

Load one chip

1. Send `0xD6 [indexLo, indexHi]`. The first `0xBC` is an acknowledgement with an empty payload.
2. Repeatedly send `0xD6 [0xAA, 0xAA]`. Append each `0xBC` payload, up to 57 bytes per reply, until exactly 128 metadata bytes are received.
3. Validate the `0xA55A` tag and parse the real code size.
4. Send `0xD7 [indexLo, indexHi]`. The first `0xBD` contains a two-byte padded transfer length.
5. Repeatedly send `0xD7 [0xAA, 0xAA]`. Append each `0xBD` payload, up to 48 bytes per reply, until the padded length is received.
6. The transfer length is $(\text{realCodeSize} + 15) \text{ AND } \text{NOT } 15$. Remove padding using the real size from metadata.
7. For string indexes 0 through 7, send `0xD8 [chipLo, chipHi, stringIndex]`. Each `0xBE` returns exactly 64 bytes; decode through the first NUL.
8. Decompile the 16-byte instruction records for editing, or retain them as raw records if the GUI is read-only.

Delete behavior

`0xDA` uses swap-last deletion. When a non-final chip is deleted, the last stored chip moves into that index and the count decreases. Therefore every cached chip index and visible order after the deleted item may be stale. Refresh the complete list after success.

`0xD3` erases all user chips and resets the count. Treat this as destructive and require an explicit confirmation in the GUI.

Images

Limits and slot policy

- Screen maximum: 160 by 128 pixels.
- Slot range: 0 through 239.
- Slot 0 is the factory logo and is hidden from normal selection.
- Slots 0 through 14 are protected defaults.
- Upload destinations are slots 1 through 239; overwriting an occupied allowed slot is possible.
- Individual deletion is allowed only for slots 15 through 239.
- Maximum compressed image data: 4096 bytes.
- Image-name field: 32 bytes; the GUI writes at most 31 UTF-8 bytes plus NUL.

RGB565 RLE format

Pixels are row-major, left to right and top to bottom. Each run is three bytes:

```
[count from 1 to 255] [RGB565 high byte] [RGB565 low byte]
```

RGB565 uses `rrrrrrgggggbbbbbb`. Split runs longer than 255. Compressed data length must be a multiple of three. When decoding, reject a zero count, truncated triplet, pixel total other than `width * height`, dimensions outside 1..160 by 1..128, or data larger than 4096 bytes.

Local editable image format

Image files are UTF-8 text. Header values appear before `[data]`; blank lines are optional. Every data line contains three decimal byte values.

```
name=Example Image
width=160
height=128
size=1230
[data]
20 248 0
15 7 224
```

`size` is the number of compressed bytes, not the number of runs or pixels. It must equal three times the number of data lines. The expanded runs must equal width times height.

List and load

1. Send `0xD0`; read the 16-bit `0xB7` count as informational.
2. Scan the displayed slot range using `0xD4 [slotLo,slotHi]`; store `0xBB` names and treat `0xB3` as empty.
3. To open a slot, send `0xE7 [slotLo,slotHi]`.
4. Parse `0xE9`: bytes 0..31 name, byte 32 width, byte 33 height, bytes 34..37 data size little-endian.
5. Starting at offset zero, send `0xE8 [slotLo,slotHi,offsetLo,offsetHi]`.
6. Each `0xEA` begins with the returned offset followed by up to 61 data bytes. Verify the offset, append data, and continue until the advertised size is reached.
7. Decode RLE and verify the exact pixel count before displaying or allowing a save.

Upload

Build a zero-filled 128-byte metadata block:

Offset	Size	Value
0	2	Zero; device writes the valid tag on commit
2	32	Name, at most 31 UTF-8 bytes plus NUL
34	1	Width
35	1	Height
36	4	Compressed byte count, little-endian
40	88	Zero/reserved

Then perform this exact sequence:

1. `0xA0 [0x80,0x00]`; wait for success.
2. `0xEC [slotLo,slotHi]`; wait for success.
3. Send the complete 128-byte metadata with `0xA1` chunks of at most 48 bytes, acknowledging each.
4. `0xA2 []`; wait for success.
5. `0xA3 [dataLengthLo,dataLengthHi]`; wait for success.
6. Send all RLE bytes with `0xA4` chunks of at most 48 bytes, acknowledging each.
7. `0xA5 []`; wait for success.
8. Refresh image slots and reload the saved slot if desired.

Delete

Send `0xEB [slotLo,slotHi]` only for 15 through 239. On success, mark the slot empty and refresh. `0xD1` erases user images while preserving default slots; require confirmation and wait through any `0xB5` responses.

Fonts

Font model and local file

A font contains fixed-width glyphs. The first glyph represents ASCII 32; later glyphs advance in order. Height is currently 1 through 8. Each glyph contains `width` vertical columns, and each column is stored as one byte. In the editable file and editor model, the most significant displayed bit is the top pixel.

The GUI permits width 1 through 32, height 1 through 8, and 1 through 224 glyphs. The device has six font slots, numbered 0 through 5. Keep uploaded data at or below the device's 4096-byte font-data area.

```
name=Example Font
width=5
height=7
size=475

[data]

# Glyph 0 - Space
0b0000000, 0b0000000, 0b0000000, 0b0000000, 0b0000000

# Glyph 1 - !
0b0000000, 0b0000000, 0b1011111, 0b0000000, 0b0000000
```

Comments after `[data]` are allowed. Parse tokens beginning with `0b`. Every token must contain exactly `height` bits. The token count must equal `size` and be divisible by `width`; glyph count is `size / width`.

List response

Send `0xDE [slot]` for every slot. `0xE1 [slot]` means empty. A `0xE2` payload is:

Offset	Size	Meaning
0	1	Slot
1	1	Width
2	1	Height
3	1	Glyph count
4	4	Data byte count, little-endian
8	32	Name field

Upload metadata and version note

The current VetusGUI transmission uses a zero-filled 128-byte metadata block with the following offsets:

Offset	Size	Current GUI value
0	2	Zero; device writes valid tag
2	32	Name, at most 31 UTF-8 bytes plus NUL
34	1	Width

Offset	Size	Current GUI value
35	1	Height
36	1	Bytes per glyph, equal to width
37	1	Glyph count
38	2	Reserved zero
40	4	Data size, little-endian
44	84	Reserved zero

Font metadata has changed during firmware development. A firmware build whose public `FontMetadata` structure instead places `dataSize` at 36, `glyphs` at 40, and `bytesEach` at 41 is a different protocol profile and must not be mixed with the current GUI profile above. Read the firmware version first and bind a tested metadata profile to that version; do not guess offsets.

Upload sequence

1. Validate name, dimensions, glyph count, and exact data length.
2. Send `0xC0 [slot]`; wait for success.
3. Send all 128 metadata bytes using `0xC1` chunks of at most 48 bytes, acknowledging each.
4. Send `0xC2 []`; wait for success.
5. Send `0xC3 [dataLengthLo,dataLengthHi]`; wait for success.
6. Convert each editable column to its low eight bits and send all bytes using `0xC4` chunks of at most 48 bytes, acknowledging each.
7. Send `0xC5 []`; wait for success.
8. Send `0xDE [slot]` and verify the returned metadata.

Delete and default font

Send `0xE6 [slot]` to delete a font. The device rejects deletion of the active/default font, so first select another populated slot through setting `0x07`, wait for the updated `0xE5`, and only then delete. Refresh all six slots after success.

UI behavior required for a complete replacement

A complete GUI should expose these capabilities:

- Port discovery, connect, disconnect, connection state, serial number, firmware version, manual refresh, bounded reconnect, and readable errors.
- Local chip creation, open, edit, Save, Save As, unsaved-change confirmation, metadata fields, eight strings, source validation, and compiled-size reporting.
- Test preparation, upload-mode check, progress by stage, retry handling, RAM execution, save-one-chip, save-all/batch, device chip list, load, edit a loaded copy, delete one, and delete all.
- Image creation/import, crop or resize to the 160 by 128 limit, pixel editing if desired, RGB565 preview, RLE validation, local text save/open, device slot list, load, upload/overwrite, and permitted deletion.
- Font creation/import, glyph bitmap editing, local text save/open, six-slot status, upload/overwrite, default-font selection, and deletion of non-active fonts.
- Settings readback and controls for temperature unit, contrast, startup logo, logo speed, default chip, LEDs, AUX/fan, default font, and Celsius/Fahrenheit minimum and maximum values.
- A transaction log that names operations and errors without dumping secrets or treating binary bytes as text.

Disable conflicting controls during an active transfer. Show both the content name and current stage, such as “Image metadata 96/128” or “Chip 3/12: code 189/320.” A Cancel action may stop the host workflow, but the next operation should begin from a fresh begin command rather than assuming partially transferred state can be resumed.

Validation checklist

Before release, verify all of the following with a real device:

1. CRC and COBS known-answer tests, including payloads containing zero bytes and frames requiring a `0xFF` COBS code.
2. Fragmented receive data, several frames in one read, an empty delimiter, a bad CRC, and an overlong unterminated frame.
3. Sequence wraparound, a late response, `0xB1` retry, failure, bad size, timeout, disconnect, and reconnect.
4. Version and serial decoding and complete startup refresh.
5. Test code with zero-valued words, registers, functions/calls, all eight strings, a 63-byte UTF-8 string boundary, several code chunks, RAM run, save, and batch save.
6. Chip load with metadata split across three responses, padded code, all eight strings, swap-last deletion, and delete-all confirmation.
7. Image RLE with a run of 255, a run longer than 255, embedded zero color bytes, maximum dimensions, near-4096-byte content, load/upload round-trip, protected-slot rejection, and user-slot deletion.
8. Font file parsing, glyph boundary validation, six-slot refresh, upload verification, default-font change, and active-font deletion rejection.
9. Every settings control, including the active-low GUI encodings for logo and LEDs, with UI state taken from the returned `0xE5` snapshot.

Implementation cautions

- Never send a native-language structure directly. Construct every field by byte offset.
- Never rely on host endianness, structure padding, signed `char`, or implicit string termination.
- Truncate by UTF-8 bytes, not characters, without cutting a multibyte sequence in half.
- Validate declared sizes before allocating or appending.
- Bind metadata-layout profiles to verified firmware versions.
- Preserve unknown local-file metadata when practical, but reject malformed binary sizes.
- Refresh collections after saves and deletes; indexes are device storage positions, not permanent identifiers.
- Keep source documents separate from compiled bytes. Local Save is not the same operation as Save to Vetus.