SiccaLab
Test & Measurement, Simple

Serial Command Reference

Firmware version 1.0

Document v1.0

Supported boards

Arduino Uno · Arduino Nano · Arduino Micro · Arduino Mega 2560

Table of Contents

1. Overview

This document is the authoritative reference for the serial command set implemented by the SiccaLab firmware running on supported Arduino boards. It describes every command the firmware recognises, the exact responses and error messages it emits, and per-board behavioural differences.

The firmware turns a supported Arduino into a serial-controlled, multi-function test and measurement device. It is intended to be driven by the SiccaLab desktop application, but every command is plain ASCII over a 115 200 baud serial link and can be exercised from any serial terminal, script, or custom host application.

1.1 Key features

- Plain-ASCII serial command set over USB serial at 115 200 baud.
- Digital I/O with per-pin and bulk operations.
- Analog input with adjustable reference voltage.
- PWM duty-cycle output and tone-based square wave generation.
- Data acquisition (DAQ) at up to 100 Hz single- or dual-channel, streamed as a binary protocol.
- SPI master and slave with persistent EEPROM-stored configuration.
- I2C master and slave with bus scan, repeated-start transactions, and a slave TX buffer.
- UART forwarding (Mega 2560 only: UART1/UART2/UART3; Arduino Micro: UART1 only).
- Debug mode for verbose acknowledgement of write/configure commands.

1.2 Resource limits by board

All limits below are extracted from the firmware source. The firmware itself does not always enforce these bounds — see the per-command sections for details.

Resource	Uno	Nano	Micro	Mega 2560
Digital pins iterated	20 (D0–D19)	22 (D0–D21)	18 (firmware-defined)	70 (D0–D69)
Analog channels	6 (A0–A5)	8 (A0–A7)	6 (A0–A5)	16 (A0–A15)
Command buffer size	80 bytes	80 bytes	80 bytes	256 bytes
SPI ring buffer	32 bytes	32 bytes	32 bytes	64 bytes
I2C ring buffer	32 bytes	32 bytes	32 bytes	32 bytes
UART channels	—	—	1 (UART1)	3 (UART1/2/3)
UART RX ring	—	—	64 bytes	256 bytes

1.3 Source of truth

This document was authored from a direct reading of the firmware source for all four boards. Where the firmware behaviour differs from prior documentation, the firmware behaviour is treated as canonical and documented as such. Section 14 (Known behaviours) lists every quirk that an experienced user might find surprising.

2. Communication Protocol

2.1 Serial settings

Parameter	Value
Baud rate	115 200
Data bits	8
Parity	None
Stop bits	1
Flow control	None
Line terminator (incoming)	`\n` or `\r`
Line terminator (outgoing)	`\n`

2.2 Command syntax

- Commands are ASCII, terminated by CR, LF, or both.
- Commands are upper-cased in place by the parser before dispatch. Send in any case.
- A space separates the command keyword from its parameters. Parameters are comma-separated where multiple are needed.
- Query commands end with a question mark (?).
- Pin and channel indices use # as the delimiter (for example `GET:DIG#5?`).
- If the input exceeds the command buffer size (see §1.2), the excess bytes are silently discarded; only the first `buffer_size - 1` bytes are parsed.

2.3 Response model

- Query responses are always emitted, regardless of debug mode.
- Write and configure commands are silent by default.
- When debug mode is on (`DEBUG:ON`), write and configure commands emit an echo line followed by `[OK]`.
- Errors are always emitted, regardless of debug mode, in the form `[ERR] <description>`.
- Unrecognised commands return `[ERR] Unknown command`.
- Some unconditional notifications (SPI config saved, DAQ stopped, EEPROM reset, restart, etc.) print regardless of debug mode — see each command for specifics.

2.4 Startup sequence

After power-on or *RESTART, the firmware:

- Disables the watchdog timer (recovery from the *RESTART path).
- Opens the serial port at 115 200 baud.
- On Arduino Micro only: waits up to 3 seconds for USB CDC enumeration. On Uno/Nano/Mega this wait is effectively instant because `Serial` is a hardware UART.
- Initialises EEPROM (checks the magic byte; writes defaults on first boot only).

- Loads the SPI configuration from EEPROM and applies it.
- Emits `READY` on a single line, then enters the command loop.

*Hosts should listen for the READY line after opening the port (or after sending *RESTART). It is the only synchronisation signal; the firmware does not provide a *OPC? query.*

3. System Commands

*IDN?

Query device identity.

Parameters	None.
Response	`SiccaLab,<model>,FW:<version>`
Boards	Uno · Nano · Micro · Mega

Example:

```
> *IDN?
SiccaLab,ArduinoMega,FW:0.1
```

*BOARD?

Query a short board identifier string.

Parameters	None.
Response	One of: `UNO | NANO | MICRO | MEGA2560`
Boards	Uno · Nano · Micro · Mega
Per-board notes	Each firmware returns its own hardcoded identifier.

*RST

Reset all I/O pins to INPUT/LOW and tear down active peripherals.

Parameters	None.
Effects	Calls SPI.end() and clears SPI buffers (in-memory only — does not touch EEPROM). Calls Wire.end() and clears I2C state. On Mega and Micro: closes every active UART channel. For every digital pin: stops tone, sets INPUT, drives LOW.
Response (default)	Silent.
Response (DEBUG:ON)	`Reset done` `[OK]`
Boards	Uno · Nano · Micro · Mega

*RESTART

Force a watchdog-driven reset of the board.

Parameters	None.
Effects	Prints "Restarting...", flushes serial, waits 200 ms, arms the watchdog at 15 ms, and spins. The MCU resets through the bootloader and re-enters setup().
Response	`Restarting...` Followed (after ~250 ms) by the normal startup READY line.
Boards	Uno · Nano · Micro · Mega

*EEPROM:RESET

Clear the EEPROM-persisted configuration (SPI mode/speed/order).

Parameters	None.
Effects	Writes 0xFF to EEPROM addresses 0x0000–0x0007 and to the magic byte at 0x00FF. On the next cold boot, EEPROMManager::begin() will re-initialise defaults. Does NOT update the in-memory SPI configuration. CONF:SPI:GET? immediately after this command will still report the previously-loaded configuration; the new state takes effect after *RESTART.
Response (default)	Silent.
Response (DEBUG:ON)	`EEPROM reset` `[OK]`
Boards	Uno · Nano · Micro · Mega

GET:ALL:PIN:MODE?

Query the configured mode of every digital pin.

Parameters	None.
Response	`D0:<mode>,D1:<mode>,...,D<N-1>:<mode>` Each <mode> is IN, OUT, or INV (invalid / no port mapping).
Boards	Uno · Nano · Micro · Mega
Per-board notes	Number of fields depends on the digital pin count for the board (see §1.2).

Example:

```
> GET:ALL:PIN:MODE?
D0:IN,D1:IN,D2:OUT,D3:IN,...,D13:OUT
```

4. Digital I/O Commands

The firmware accepts any pin index supplied by the host; it does not validate against the board's physical pin count for write or query operations on a single pin. Out-of-range pins fall through to the Arduino core, which silently no-ops. Only the multi-pin query (GET:DIG:MULTIPLE) enforces a range check.

4.1 Configuration

CONF:DIG:INPUT#<n>

Set a single pin to INPUT mode.

Parameters	n — digital pin index (no range check).
Effects	pinMode(n, INPUT)
Response (default)	Silent.
Response (DEBUG:ON)	`D<n> INPUT` `[OK]`
Errors	None.
Boards	Uno · Nano · Micro · Mega

CONF:DIG:OUTPUT#<n>

Set a single pin to OUTPUT mode.

Parameters	n — digital pin index.
Effects	pinMode(n, OUTPUT)
Response (DEBUG:ON)	`D<n> OUTPUT` `[OK]`
Boards	Uno · Nano · Micro · Mega

CONF:DIG:INPUT:ALL

Set every digital pin on the board to INPUT mode.

Parameters	None.
Effects	pinMode(pin, INPUT) for pin = 0 ... pin_count − 1 (see §1.2).
Response (DEBUG:ON)	`All INPUT` `[OK]`
Boards	Uno · Nano · Micro · Mega

CONF:DIG:OUTPUT:ALL

Set every digital pin to OUTPUT mode.

Parameters	None.
Effects	pinMode(pin, OUTPUT) for all pins.
Response (DEBUG:ON)	`All OUTPUT` `[OK]`

Boards	Uno · Nano · Micro · Mega

4.2 Write

`SET:DIG#<n> <value>`

Write a logical level to a single pin.

Parameters	n — pin index. value — interpreted with atoi(): non-zero → HIGH, zero or non-numeric → LOW. No range validation.
Effects	digitalWrite(n, value)
Response (DEBUG:ON)	`D<n>=<value>` `[OK]`
Boards	Uno · Nano · Micro · Mega

`SET:DIG:ALL <value>`

Write the same level to every digital pin.

Parameters	value — as above.
Response (DEBUG:ON)	`All=<value>` `[OK]`
Boards	Uno · Nano · Micro · Mega

4.3 Read

`GET:DIG#<n>?`

Read the current level of a single pin.

Parameters	n — pin index.

Response	`0`
	or
	`1`

Boards	Uno · Nano · Micro · Mega

`GET:DIG:ALL?`

Read every digital pin.

Parameters	None.
Response	`D0:<v>,D1:<v>,...,D<N-1>:<v>` Number of fields = digital pin count for the board (see §1.2).
Boards	Uno · Nano · Micro · Mega

`GET:DIG:MULTIPLE [#]<p1>,<p2>[,...][?]`

Read a comma-separated list of pins. An optional leading # and trailing ? are stripped by the parser.

Parameters	Comma-separated pin list.
Validation	Each pin is checked against the digital pin count. The first invalid pin halts the

	operation.
Response	`D<p1>:<v>,D<p2>:<v>,...`
Errors	`[ERR] Need pin list` — when no parameters were supplied. `[ERR] Invalid pin` — when any parsed pin is out of range.
Boards	Uno · Nano · Micro · Mega

Example:

```
> GET:DIG:MULTIPLE #2,5,7,12?
D2:1,D5:0,D7:1,D12:0
```

5. Analog Input Commands

Analog channels are addressed by their A-prefixed index (A0, A1, …). The reference voltage is software-defined for ADC-to-volts conversion only; it does not change the AVR's analog reference register.

CONF:ANALOG:VREF [<value>]

Set or query the reference voltage used by the firmware to scale ADC readings to volts.

Parameters	value — float, 0.1 ≤ value ≤ 5.0. Omit the parameter to query the current value.
Default	5.0 (in-memory only; not persisted to EEPROM).
Response (query)	`<value>` Example: 5.00
Response (set, DEBUG:ON)	`<value>` `[OK]`
Errors	`[ERR] Range 0.1-5.0V` — when set value is out of range.
Boards	Uno · Nano · Micro · Mega

GET:ANALOG#<n>?

Read a single analog channel.

Parameters	n — analog channel index. Validated against the analog channel count for the board (see §1.2).
Response	`<volts>` Three decimal places, computed as: volts = analogRead(A0 + n) × (VREF / 1023.0).
Errors	`[ERR] Invalid channel` — when n is out of range for the board.
Boards	Uno · Nano · Micro · Mega

GET:ANALOG:ALL?

Read every analog channel.

Parameters	None.
Response	`A0:<v>,A1:<v>,...,A<N-1>:<v>`
Boards	Uno · Nano · Micro · Mega

GET:ANALOG:MULTIPLE [#]<c1>,<c2>[,...][?]

Read a comma-separated list of analog channels.

Parameters	Comma-separated channel list (optional leading # and trailing ? stripped).
Validation	Each channel checked against the board's analog channel count.
Response	`A<c1>:<v>,A<c2>:<v>,...`
Errors	`[ERR] Need channel list` — when no parameters were supplied.

	`[ERR] Invalid channel` — when any channel is out of range.
Boards	Uno · Nano · Micro · Mega

6. PWM & Signal Generation

The firmware exposes Arduino's analogWrite() for duty-cycle PWM and the Arduino tone() library for frequency-based square wave generation. It does not validate PWM-capable pins; on non-PWM pins the underlying Arduino call silently no-ops.

SET:PWM <pin>,<duty>

Generate a duty-cycled PWM signal on a pin.

Parameters	pin — digital pin index. duty — integer 0–100 (percent). Values are constrained to 0–100, then mapped to 0–255 internally.
Effects	pinMode(pin, OUTPUT); analogWrite(pin, mapped)
Response (DEBUG:ON)	`PWM D<pin> @<duty>%` `[OK]`
Errors	`[ERR] Need pin,duty` — when fewer than two comma-separated parameters are supplied.
Boards	Uno · Nano · Micro · Mega
Notes	No PWM-capability check. analogWrite on a non-PWM pin is a no-op at the AVR core level.

SET:SQUARE:ON <pin>,<freq>

Start a square wave on a pin at the specified frequency.

Parameters	pin — digital pin index. freq — unsigned long, Hz. No range check in the firmware.
Effects	pinMode(pin, OUTPUT); tone(pin, freq)
Response (DEBUG:ON)	`Square D<pin> @<freq>Hz` `[OK]`
Errors	`[ERR] Need pin,freq`
Boards	Uno · Nano · Micro · Mega
Notes	Arduino's tone() uses Timer2 on Uno/Nano and Timer2 by default on Mega. Below ~31 Hz, tone() may not produce reliable output.

SET:SQUARE:OFF <pin>

Stop the square wave on a pin and return it to INPUT.

Parameters	pin — digital pin index.
Effects	noTone(pin); pinMode(pin, INPUT)
Response (DEBUG:ON)	`Square OFF D<pin>` `[OK]`
Errors	`[ERR] Need pin`
Boards	Uno · Nano · Micro · Mega

7. Data Acquisition (DAQ)

The DAQ subsystem provides single- or dual-channel ADC sampling at up to 100 Hz. Samples are streamed as a 6-byte binary protocol from the moment DAQ:START is accepted until DAQ:STOP is received or the timeout expires. The output stream is binary; do not display it as text.

7.1 Commands

`DAQ:START <ch1>[,<ch2>],<freq>`

Begin streaming samples.

Parameters	ch1 — first analog channel. Constrained to 0 … (analog channel count − 1) for the board. ch2 — optional second channel. Same constraint. freq — sample rate in Hz, constrained 1 ≤ freq ≤ 100.
Effects	Arms Timer1 in CTC mode at the chosen frequency. Sampling proceeds in the timer ISR; packets are flushed in the main loop.
Response (DEBUG:ON, single channel)	`DAQ A<ch1> @<freq>Hz` `[OK]`
Response (DEBUG:ON, dual channel)	`DAQ A<ch1>,A<ch2> @<freq>Hz` `[OK]`
Errors	`[ERR] Need ch1[,ch2],freq`
Boards	Uno · Nano · Micro · Mega
Notes	Timer1 prescaler is selected automatically: 256 for freq ≤ 4 Hz, 64 for freq ≤ 30 Hz, 8 for freq > 30 Hz. Hardcoded for 16 MHz F_CPU.

`DAQ:STOP`

Stop streaming.

Parameters	None.
Effects	Disables Timer1 and stops the packet stream. Does NOT emit a sentinel packet.
Response	`DAQ stopped` Always printed, regardless of debug mode.
Boards	Uno · Nano · Micro · Mega

`DAQ:TIMEOUT [<seconds>]`

Set or query the automatic stop timeout for a DAQ run.

Parameters	seconds — integer, valid range 1–1800. Omit to query.
Default	1800 seconds (30 minutes).
Response (query)	`<seconds>s`
Response (set, DEBUG:ON)	`<seconds>s` `[OK]`

Errors	`[ERR] Range 1-1800s`
Boards	Uno · Nano · Micro · Mega

DAQ:TIMEOUT?

Query the current timeout value.

Parameters	None.
Response	`<seconds>s`
Boards	Uno · Nano · Micro · Mega

7.2 Binary packet format

Each sample is transmitted as a fixed 6-byte packet over the serial port. Both single-channel and dual-channel runs use the same 6-byte structure; in single-channel mode, the channel-2 bytes are sentinels.

Offset	Byte	Meaning
0	0xAA	Start-of-packet marker.
1	`val1 & 0xFF`	Channel 1 ADC low byte (bits 0–7).
2	`(val1 >> 8) & 0x03`	Channel 1 ADC high 2 bits (bits 8–9). Upper 6 bits zero.
3	`val2 & 0xFF  or  0xFF`	Channel 2 ADC low byte, OR 0xFF sentinel in single-channel mode.
4	`(val2 >> 8) & 0x03  or  0xFF`	Channel 2 ADC high 2 bits, OR 0xFF sentinel.
5	0x55	End-of-packet marker.

Reconstructing an ADC value:

```
value = (byte_high << 8) | byte_low      // 10-bit, 0..1023
volts = value * (VREF / 1023.0)          // host must remember VREF
```

Timeout sentinel:

When the DAQ timeout expires, the firmware emits a single packet of six 0xFF bytes, then stops sampling. This is the ONLY end-of-stream signal — DAQ:STOP does not emit it.

```
0xFF 0xFF 0xFF 0xFF 0xFF 0xFF      // timeout-expired packet
```

Host parser notes:

- Lock onto the 0xAA start marker; validate the 0x55 end marker.
- The VREF used at sampling time is not transmitted in the packet — your host must keep track of the most recent CONF:ANALOG:VREF setting.
- If you see ASCII text mixed into the stream, you have not stopped DAQ; the only ASCII output during a DAQ run is the response to a subsequent command (e.g. DAQ:STOP itself returning "DAQ stopped").
- In single-channel mode, the channel-2 sentinel pattern is bytes `0xFF 0xFF` at offsets 3 and 4. A genuine dual-channel sample with value 0x03FF would have offset 4 equal to `0x03` — never 0xFF — so the patterns are unambiguous.

8. SPI Interface

The SPI controller backs a single hardware SPI peripheral. The active mode (off / master / slave) is persisted to EEPROM. Reconfiguration of master/slave mode writes EEPROM and prompts the host to issue *RESTART; the new mode takes effect only after the board reboots. CONF:SPI:DISABLE is the exception — it takes effect immediately in addition to clearing EEPROM.

8.1 Speed and mode reference

Speed index (master mode):

Index	Speed
0	4 MHz
1	2 MHz
2	1 MHz
3	500 kHz
4	250 kHz
5	125 kHz

SPI modes:

Mode	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

Bit order:

Value	Order
0	MSB first
1	LSB first

8.2 Configuration

`CONF:SPI:MASTER <speed>,<mode>,<order>`

Configure SPI as a master and persist the settings to EEPROM.

Parameters	speed — speed index 0–5 (see §8.1). mode — SPI mode 0–3. order — 0 = MSB first, 1 = LSB first.
Effects	Writes the configuration to EEPROM. The new mode takes effect only after *RESTART. Until then, the previously-loaded mode remains active.
Response	`SPI Master saved. *RESTART` Always printed, regardless of debug mode.
Response	`[OK]`

(DEBUG:ON)	
Errors	`[ERR] Need spd,mode,ord`
Boards	Uno · Nano · Micro · Mega
Notes	Parameter values are not range-checked at the parse step; they are clamped on the next boot when loadFromEEPROM runs.

`CONF:SPI:SLAVE <mode>,<order>`

Configure SPI as a slave and persist to EEPROM.

Parameters	mode — 0–3. order — 0 = MSB first, 1 = LSB first. Speed is irrelevant in slave mode (the master clocks the bus).
Response	`SPI Slave saved. *RESTART`
Response (DEBUG:ON)	`[OK]`
Errors	`[ERR] Need mode,ord`
Boards	Uno · Nano · Micro · Mega

`CONF:SPI:DISABLE`

Disable SPI and clear the persisted configuration. Takes effect immediately.

Parameters	None.
Effects	Writes 'disabled' to EEPROM. Calls SPI.end(). Clears RX/TX ring buffers. Sets in-memory mode to disabled (unlike Master/Slave config, no *RESTART needed).
Response	`SPI disabled`
Response (DEBUG:ON)	`[OK]`
Boards	Uno · Nano · Micro · Mega

`CONF:SPI:GET?`

Query the current SPI configuration.

Parameters	None.

Response	One of:
	`SPI:OFF`
	`SPI:MASTER,<speed>,<mode>,<order>`
	`SPI:SLAVE,<mode>,<order>`

Boards	Uno · Nano · Micro · Mega
Notes	Reports the currently-active configuration. If you have just issued CONF:SPI:MASTER or CONF:SPI:SLAVE but not yet sent *RESTART, this query still reports the OLD configuration.

8.3 Data transfer

SPI:SEND <cs_pin>,<cs_active>,<byte1>[,<byte2>,...]

Send one or more bytes to an SPI slave. Master mode only.

Parameters	cs_pin — digital pin used as chip-select. cs_active — 0 for active-LOW, 1 for active-HIGH. byte1, ... — bytes to transfer. Each parsed as plain hex (no 0x prefix needed; values are always hex).
Effects	pinMode(cs_pin, OUTPUT) SPI.beginTransaction(<configured settings>) Drives CS to its inactive level, then to active. Transfers each byte via SPI.transfer; stores each received byte in the RX ring buffer. Drives CS back to inactive. SPI.endTransaction()
Response (DEBUG:ON)	`Sent <n> CS:<cs_pin>` `[OK]`

Errors	`[ERR] Not Master`
	— SPI not currently in master mode.
	`[ERR] Need cs,act,data`

Boards	Uno · Nano · Micro · Mega
Buffer limit	Maximum data length is bounded by the command buffer size (§1.2) and the SPI ring buffer (32 bytes on Uno/Nano/Micro, 64 on Mega).

SPI:REQUEST <cs_pin>,<cs_active>,<num>

Clock out num dummy bytes and capture the returned bytes into the SPI RX buffer. Master mode only.

Parameters	cs_pin — chip-select pin. cs_active — 0 or 1. num — 8-bit value, 0–255. The firmware does NOT bound num against the SPI buffer size.
Effects	Clocks out num bytes of 0x00 with CS held active. Each returned byte is pushed into the SPI RX ring buffer.
Overflow behaviour	If num exceeds the SPI ring buffer size (32 / 32 / 32 / 64 for Uno / Nano / Micro / Mega), older bytes are dropped and the OVF flag is appended to the next SPI:READ? response.
Response (DEBUG:ON)	`[OK]`
Errors	`[ERR] Not Master` `[ERR] Need cs,act,num`
Boards	Uno · Nano · Micro · Mega

SPI:READ?

Drain the SPI RX ring buffer.

Parameters	None.
Response	Empty line if the buffer is empty, otherwise:

	`0xNN,0xNN,...[,OVF]` The trailing ,OVF token is appended if the ring overflowed since the last read.
Boards	Uno · Nano · Micro · Mega

SPI:SENT?

Slave mode: query how many bytes the slave returned in the last master-driven transfer.

Parameters	None.
Response	`<count>`
Errors	`[ERR] Not Slave`
Boards	Uno · Nano · Micro · Mega

SPI:AVAILABLE?

Query the number of bytes currently in the SPI RX ring.

Parameters	None.
Response	`<count>`
Boards	Uno · Nano · Micro · Mega

SPI:CLEAR

Clear the SPI RX/TX ring buffers and the overflow flag.

Parameters	None.
Response (DEBUG:ON)	`[OK]`
Boards	Uno · Nano · Micro · Mega

9. I2C Interface

The I2C controller wraps the AVR Wire library. Mode (off / master / slave) lives in RAM only — it is NOT persisted to EEPROM, so the bus is closed on every cold boot and after every *RESTART. Master operations support single-shot read, write, and combined write/read with a repeated start. Slave operations expose received bytes via a 32-byte RX ring and a single 31-byte TX scratch buffer.

Important — address parsing:

All I2C address and data byte parameters are parsed as HEX, with or without a 0x prefix. CONF:I2C:SLAVE 18 is interpreted as 0x18 (decimal 24), not decimal 18. CONF:I2C:SLAVE 0x42 is also valid and is interpreted as 0x42 (decimal 66). This applies to addresses in CONF:I2C:SLAVE, I2C:WRITE, I2C:READ, I2C:WRITEREAD, and to every data byte.

9.1 Speed indices

Index	Speed
0	100 kHz (standard mode)
1	400 kHz (fast mode)

9.2 Configuration

`CONF:I2C:MASTER <speed>`

Configure the I2C peripheral as a master.

Parameters	speed — index 0 (100 kHz) or 1 (400 kHz).
Effects	Wire.begin(); Wire.setClock(speed_hz)
Response (DEBUG:ON)	`I2C Master <speed_hz>` `[OK]`
Errors	`[ERR] Need speed` `[ERR] Invalid speed`
Boards	Uno · Nano · Micro · Mega

`CONF:I2C:SLAVE <addr>`

Configure the I2C peripheral as a slave at the specified address. Address is always parsed as HEX (see §9 introduction).

Parameters	addr — 7-bit slave address, parsed as hex. Valid range: 1–127 (i.e. 0x01–0x7F).
Effects	Registers the slave address with Wire. Hooks the onReceive and onRequest ISRs. Initialises a 31-byte TX scratch and a 32-byte RX ring.
Response (DEBUG:ON)	`I2C Slave 0x<addr>` `[OK]`
Errors	`[ERR] Need addr` `[ERR] Invalid addr (1-127)`
Boards	Uno · Nano · Micro · Mega

CONF:I2C:DISABLE

Disable the I2C peripheral.

Parameters	None.
Effects	Wire.end(); clears RX/TX state; mode set to off.
Response (DEBUG:ON)	`I2C disabled` `[OK]`
Boards	Uno · Nano · Micro · Mega

CONF:I2C:GET?

Query the current I2C configuration.

Parameters	None.

Response	One of:
	`I2C:OFF`
	`I2C:MASTER,<speed_index>`
	`I2C:SLAVE,0x<addr>`

Boards	Uno · Nano · Micro · Mega

9.3 Master mode

I2C:SCAN?

Scan the bus by attempting a zero-byte write to every 7-bit address.

Parameters	None.
Response	Semicolon-separated list of responding addresses: `0xNN;0xNN;...` If no devices respond, no output is produced (not even a newline).
Errors	`[ERR] Not Master`
Boards	Uno · Nano · Micro · Mega

I2C:WRITE <addr>,<byte1>[,<byte2>,...]

Write one or more bytes to an I2C slave. Both addr and data bytes are parsed as hex.

Parameters	addr — slave address (hex), 1–127. byte1, ... — data bytes (hex).
Effects	Wire.beginTransmission(addr); writes each byte; Wire.endTransmission().
Response (DEBUG:ON, success)	`[OK]`

Errors	`[ERR] Not Master`
	`[ERR] Need addr,data`
	`[ERR] Invalid addr`
	On Wire error, two lines are emitted:
	`I2C Error <code>`
	`[ERR] Write failed`

Boards	Uno · Nano · Micro · Mega
Wire error	1 = data too long, 2 = NACK on address, 3 = NACK on data, 4 = other, 5 =

codes	timeout.

`I2C:READ <addr>,<num>`

Read num bytes from an I2C slave.

Parameters	addr — slave address (hex), 1–127. num — decimal, 1–32.
Response	If bytes are received: `0xNN,0xNN,...` If zero bytes are received, no output is emitted (not even a newline). See §14 for the rationale.

Errors	`[ERR] Not Master`
	`[ERR] Need addr,num`
	`[ERR] Invalid addr`
	`[ERR] Invalid num (1-32)`

Boards	Uno · Nano · Micro · Mega

`I2C:WRITEREAD <addr>,<num>,<byte1>[,<byte2>,...]`

Write one or more bytes, then read num bytes back without releasing the bus (repeated start). Typical for register reads.

Parameters	addr — hex, 1–127. num — decimal, 1–32 (bytes to read). byte1, ... — hex bytes to write first.
Response	If bytes are received: `0xNN,0xNN,...`

Errors	`[ERR] Not Master`
	`[ERR] Need addr,num,data`
	`[ERR] Invalid addr`
	`[ERR] Invalid num (1-32)`
	`[ERR] Write failed`
	`[ERR] No data`

Boards	Uno · Nano · Micro · Mega

9.4 Slave mode

`I2C:RECEIVED?`

Read out any bytes the slave has received from the master since the last call. Drains the slave RX ring.

Parameters	None.
Response	Empty line if the buffer is empty, otherwise: `0xNN,0xNN,...[,OVF]`
Boards	Uno · Nano · Micro · Mega
Notes	This command does NOT check that the firmware is in slave mode — if called in master or off mode, it will simply return an empty line. The RX ring is 32 bytes; bytes beyond that are dropped and the OVF flag is set.

I2C:LOADTX <byte1>[,<byte2>,...]

Load up to 31 bytes into the slave's TX scratch. These bytes will be returned the next time the master reads from this slave. Slave mode only.

Parameters	Comma-separated hex bytes.
Effects	Clears the previous TX scratch and writes the new bytes. Each LOADTX REPLACES the previous payload.
Response (DEBUG:ON)	`[OK]`
Errors	`[ERR] Not Slave` `[ERR] Need data`
Boards	Uno · Nano · Micro · Mega
Notes	Practical capacity is 31 bytes (ring buffer with one reserved slot). On Uno/Nano/Micro the command-buffer scratch limits a single I2C:LOADTX line; on Mega the line can be longer.

I2C:SENT?

Slave: query how many bytes were sent in the most recent master-read transaction. Reading this command clears the sent flag.

Parameters	None.
Response	`<count>` Zero is reported when no transfer has occurred since the last query.
Errors	`[ERR] Not Slave`
Boards	Uno · Nano · Micro · Mega

I2C:LASTTX?

Slave: query the bytes that were most recently transmitted to the master.

Parameters	None.
Response	Empty line if no transfer yet, otherwise: `0xNN,0xNN,...`
Errors	`[ERR] Not Slave`
Boards	Uno · Nano · Micro · Mega

I2C:AVAILABLE?

Query the number of bytes currently in the slave RX ring.

Parameters	None.
Response	`<count>`
Boards	Uno · Nano · Micro · Mega
Notes	Like I2C:RECEIVED?, this command has no mode check.

I2C:CLEAR

Clear both RX and TX state for the I2C peripheral.

Parameters	None.

Response	`I2C Buffers cleared`
Response (DEBUG:ON)	`[OK]`
Boards	Uno · Nano · Micro · Mega

10. UART Interface

UART forwarding is available on the Mega 2560 (three channels) and on the Arduino Micro (one channel). It is NOT compiled into the Uno or Nano firmwares — sending any UART command to those boards returns [ERR] Unknown command. The pattern <n> below stands for the channel number; valid values are 1, 2, or 3 on Mega and 1 only on Micro.

10.1 Hardware mapping

Board	UART1	UART2	UART3
Uno	—	—	—
Nano	—	—	—
Micro	Serial1 (D0 RX, D1 TX)	—	—
Mega 2560	Serial1 (D18 TX, D19 RX)	Serial2 (D16 TX, D17 RX)	Serial3 (D14 TX, D15 RX)

10.2 Configuration

CONF:UART<n>:BEGIN <baud>

Open a UART channel at the specified baud rate.

Parameters	n — channel number (1 on Micro; 1, 2, or 3 on Mega). baud — one of: 300, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200.
Effects	If the channel was already open, calls end() first. Then begin(baud) on the corresponding hardware port and clears the channel RX ring.
Response (DEBUG:ON)	`[OK]`
Errors	`[ERR] Invalid UART` — channel index out of range. `[ERR] Need baud` `[ERR] Invalid baud` — unsupported baud rate.
Boards	Micro (channel 1), Mega 2560 (channels 1, 2, 3).

CONF:UART<n>:END

Close a UART channel.

Parameters	n — channel number.
Effects	If active, calls end() and clears the RX ring.
Response (DEBUG:ON)	`[OK]`
Errors	`[ERR] Invalid UART`
Boards	Micro, Mega 2560.

CONF:UART<n>:GET?

Query the current state of a UART channel.

Parameters	n — channel number.

Response	If open:
	`UART<n>:<baud>`
	If closed:
	`UART<n>:OFF`

Errors	`[ERR] Invalid UART`
Boards	Micro, Mega 2560.

10.3 Data transfer

`UART<n>:WRITE <byte1>[,<byte2>,...]`

Send one or more bytes out a UART channel.

Parameters	Comma-separated hex bytes (no 0x prefix needed).
Effects	Writes each byte to the channel's hardware port.
Response (DEBUG:ON)	`[OK]`

Errors	`[ERR] Invalid UART`
	`[ERR] UART not open`
	`[ERR] Need data`

Boards	Micro, Mega 2560.
Per-board limits	The command-buffer size (§1.2) bounds the maximum hex payload per WRITE: ~22 bytes per line on Micro, ~81 bytes per line on Mega.

`UART<n>:READ?`

Drain the channel's RX ring buffer.

Parameters	n — channel number.
Response	Empty line if empty, otherwise: `0xNN,0xNN,...[,OVF]`
Errors	`[ERR] Invalid UART`
Boards	Micro, Mega 2560.
Notes	The firmware pulls bytes from the hardware UART into the channel ring on every main-loop tick. Ring size: 64 bytes on Micro, 256 bytes on Mega.

`UART<n>:AVAILABLE?`

Query bytes currently in the channel RX ring.

Parameters	n — channel number.
Response	`<count>`
Errors	`[ERR] Invalid UART`
Boards	Micro, Mega 2560.

`UART<n>:CLEAR`

Clear the channel RX ring and overflow flag.

Parameters	n — channel number.
Response (DEBUG:ON)	`[OK]`
Errors	`[ERR] Invalid UART`
Boards	Micro, Mega 2560.

11. Debug Mode

Debug mode controls verbosity for write and configure commands. Queries always emit their result; errors always emit [ERR] regardless. Debug mode is not persisted to EEPROM — it resets to OFF on every power-on and *RESTART.

DEBUG:ON

Enable echoing of write/configure command acknowledgements.

Parameters	None.
Response	`Debug ON`
Boards	Uno · Nano · Micro · Mega

DEBUG:OFF

Disable acknowledgement echoes (the default state).

Parameters	None.
Response	`Debug OFF`
Boards	Uno · Nano · Micro · Mega

Effect of debug mode:

- In DEBUG:OFF (default), write and configure commands are silent.
- In DEBUG:ON, write and configure commands print a short echo (the exact text is listed under each command), followed by [OK].
- Query responses are unaffected. They are always emitted.
- Errors are unaffected. They are always emitted with the [ERR] prefix.
- A subset of unconditional messages prints regardless of debug mode: "SPI Master saved. *RESTART", "SPI Slave saved. *RESTART", "SPI disabled", "I2C Buffers cleared", "DAQ stopped", and "Restarting...".

12. Error Handling

All errors are emitted as a single line with the [ERR] prefix. The list below collates every error string the firmware can produce. Where a string is board-specific, the affected boards are noted.

Error message	Triggered by
`[ERR] Unknown command`	Command not recognised by the parser.
`[ERR] Need pin list`	GET:DIG:MULTIPLE with no parameters.
`[ERR] Invalid pin`	GET:DIG:MULTIPLE with an out-of-range pin.
`[ERR] Range 0.1-5.0V`	CONF:ANALOG:VREF with value outside 0.1–5.0.
`[ERR] Invalid channel`	Analog read with channel out of range.
`[ERR] Need channel list`	GET:ANALOG:MULTIPLE with no parameters.
`[ERR] Need pin,duty`	SET:PWM missing parameters.
`[ERR] Need pin,freq`	SET:SQUARE:ON missing parameters.
`[ERR] Need pin`	SET:SQUARE:OFF missing parameter.
`[ERR] Need ch1[,ch2],freq`	DAQ:START missing parameters.
`[ERR] Range 1-1800s`	DAQ:TIMEOUT out of range.
`[ERR] Need spd,mode,ord`	CONF:SPI:MASTER missing parameters.
`[ERR] Need mode,ord`	CONF:SPI:SLAVE missing parameters.
`[ERR] Need cs,act,data`	SPI:SEND missing parameters.
`[ERR] Need cs,act,num`	SPI:REQUEST missing parameters.
`[ERR] Not Master`	SPI/I2C operation requiring master mode while in slave or off mode.
`[ERR] Not Slave`	SPI/I2C operation requiring slave mode while in master or off mode.
`[ERR] Need speed`	CONF:I2C:MASTER missing parameter.
`[ERR] Invalid speed`	CONF:I2C:MASTER with unsupported speed.
`[ERR] Need addr`	CONF:I2C:SLAVE missing parameter.
`[ERR] Invalid addr (1-127)`	CONF:I2C:SLAVE with address 0 or > 127.
`[ERR] Need addr,data`	I2C:WRITE missing parameters.
`[ERR] Need addr,num`	I2C:READ missing parameters.
`[ERR] Need addr,num,data`	I2C:WRITEREAD missing parameters.
`[ERR] Invalid addr`	I2C address 0 or > 127.
`[ERR] Invalid num (1-32)`	I2C:READ or I2C:WRITEREAD with num out of range.
`I2C Error <code>`	I2C Wire-level error preamble (followed by [ERR] Write failed).
`[ERR] Write failed`	I2C transmission ended with non-zero Wire error code.
`[ERR] No data`	I2C:WRITEREAD: master received zero bytes from the slave.
`[ERR] Need data`	I2C:LOADTX or UART<n>:WRITE missing parameters.

Error message	Triggered by
`[ERR] Invalid UART`	UART command with channel index out of range for the board.
`[ERR] UART not open`	UART<n>:WRITE on a closed channel.
`[ERR] Need baud`	CONF:UART<n>:BEGIN missing baud.
`[ERR] Invalid baud`	CONF:UART<n>:BEGIN with unsupported baud rate.

13. Examples

The examples below are written as a host might exchange them with the firmware. Lines prefixed with > are sent to the board; other lines are received.

13.1 Identification and reset

```
> *IDN?
SiccaLab,ArduinoMega,FW:0.1

> *BOARD?
MEGA2560

> *RST
```

13.2 Basic LED control

```
> CONF:DIG:OUTPUT#13
> SET:DIG#13 1                  // LED on
> SET:DIG#13 0                  // LED off
```

13.3 Reading an analog sensor

```
> CONF:ANALOG:VREF 3.3
> GET:ANALOG#0?
2.456

> GET:ANALOG:ALL?
A0:2.456,A1:0.123,A2:3.789,A3:1.567,A4:2.234,A5:0.891
```

13.4 PWM motor control

```
> SET:PWM 9,50                  // 50% duty cycle
> SET:PWM 9,75                  // increase to 75%
> SET:PWM 9,0                   // stop
```

13.5 Square wave generation

```
> SET:SQUARE:ON 8,1000          // 1 kHz on pin 8
> SET:SQUARE:OFF 8
```

13.6 Reading multiple pins / channels in one round-trip

```
> GET:DIG:MULTIPLE #2,5,7,12?
D2:1,D5:0,D7:1,D12:0

> GET:ANALOG:MULTIPLE #0,3,5?
A0:2.456,A3:1.567,A5:0.891
```

13.7 Data acquisition

```
> DAQ:TIMEOUT 60                // 60 second auto-stop
> DAQ:START 0,50                // channel 0 @ 50 Hz
<binary 6-byte packets stream>
```

```
> DAQ:STOP
DAQ stopped
```

See §7.2 for the binary packet structure.

13.8 SPI: configure and read from a sensor

```
> CONF:SPI:MASTER 0,0,0         // 4 MHz, mode 0, MSB first
SPI Master saved. *RESTART
> *RESTART
Restarting...
READY

> SPI:SEND 10,0,80              // CS=D10 active-low, send 0x80
> SPI:REQUEST 10,0,4            // clock out 4 dummy bytes
> SPI:READ?
0x12,0x34,0x56,0x78
```

13.9 I2C: scan and talk to a device

```
> CONF:I2C:MASTER 1             // 400 kHz
> I2C:SCAN?
0x27;0x68;

> I2C:WRITE 68,6B,00            // wake an MPU-6050 at addr 0x68

> I2C:WRITEREAD 68,6,3B         // read 6 bytes starting from reg 0x3B
0x12,0x34,0x56,0x78,0x9A,0xBC
```

13.10 I2C slave mode

```
> CONF:I2C:SLAVE 42             // slave address 0x42
> I2C:LOADTX AA,BB,CC,DD

// after the master reads:
> I2C:SENT?
4
> I2C:LASTTX?
0xAA,0xBB,0xCC,0xDD

// when the master writes:
> I2C:AVAILABLE?
3
> I2C:RECEIVED?
0x01,0x02,0x03
```

13.11 UART forwarding on Mega

```
> CONF:UART1:BEGIN 9600
> CONF:UART1:GET?
UART1:9600

> UART1:WRITE 48,65,6C,6C,6F    // "Hello" in hex bytes

> UART1:AVAILABLE?
```

```
5
> UART1:READ?
0x48,0x69,0x21,0x0D,0x0A

> CONF:UART1:END
```

13.12 Using debug mode during development

```
> DEBUG:ON
Debug ON

> SET:DIG#13 1
D13=1
[OK]

> DEBUG:OFF
Debug OFF

> SET:DIG#13 0                    // no echo
```

14. Known Behaviours

This section documents firmware behaviours that may differ from naive expectations. They are not bugs in need of immediate fixes; they are the current, intentional or accepted state of the firmware. They are listed here for completeness and to save host developers from guessing.

14.1 Nano returns "UNO" to *BOARD? in earlier builds

In the firmware as of this revision, each board returns its own dedicated string (NANO on the Nano, UNO on the Uno, MEGA2560 on the Mega, MICRO on the Micro). Earlier internal builds used an MCU-defined dispatch that caused Nano boards to return UNO because both share the ATmega328P MCU define. Hosts that need to distinguish Nano from Uno should rely on *IDN? (FW_MODEL field), which is always per-board.

14.2 *EEPROM:RESET does not update in-memory state

*EEPROM:RESET writes 0xFF to the SPI configuration slots in EEPROM but does NOT reload the in-memory SPI state. A CONF:SPI:GET? issued immediately after *EEPROM:RESET will still report the configuration loaded at boot. The reset takes effect on the next cold boot or *RESTART.

14.3 SPI master/slave reconfiguration requires *RESTART

CONF:SPI:MASTER and CONF:SPI:SLAVE write the new configuration to EEPROM but do not apply it immediately. Until you send *RESTART, the firmware operates in whatever SPI mode was in force at the last boot. The "SPI Master saved. *RESTART" / "SPI Slave saved. *RESTART" response is a reminder to issue *RESTART. CONF:SPI:DISABLE is the exception — it applies immediately.

14.4 SPI:REQUEST has no upper bound on num

The num parameter is parsed as an unsigned 8-bit value (0–255) and not validated against the SPI ring buffer size. Requesting more bytes than the ring buffer (32 on Uno/Nano/Micro, 64 on Mega) causes earlier bytes to be discarded and the OVF flag to be appended to the next SPI:READ? response. Keep num within the buffer size to avoid losing data.

14.5 I2C addresses and data are always hex

Every I2C address and data byte parameter is parsed with strtol(…, 16). This is the firmware's intentional protocol convention. CONF:I2C:SLAVE 18 is interpreted as 0x18 = decimal 24, not decimal 18. To set decimal address 18, send CONF:I2C:SLAVE 12 (0x12 = decimal 18) or CONF:I2C:SLAVE 0x12. The same rule applies to addresses in I2C:WRITE, I2C:READ, I2C:WRITEREAD, and to all data byte parameters.

14.6 I2C:RECEIVED? and I2C:AVAILABLE? have no mode check

These two queries execute even when I2C is configured as a master or is disabled. They will simply return an empty buffer in those modes. Only I2C:LOADTX, I2C:SENT?, and I2C:LASTTX? enforce slave mode and respond with [ERR] Not Slave otherwise.

14.7 I2C:WRITE failure emits two lines

On a Wire-level error, the firmware emits both an "I2C Error <code>" line and an "[ERR] Write failed" line. Hosts that poll for exactly one [ERR] line per failed command should be prepared for this two-line pattern.

14.8 I2C:READ emits nothing on zero bytes received

If the slave returned no data, I2C:READ produces no output at all — not even an empty newline. Host code cannot distinguish "command accepted, no data" from "firmware never

responded" without applying a timeout. I2C:WRITEREAD, by contrast, returns the explicit error [ERR] No data in the same situation.

14.9 DAQ:STOP does not emit a sentinel

Only the DAQ timeout path emits the six-0xFF end-of-stream packet. DAQ:STOP simply stops sampling and replies with the ASCII line "DAQ stopped". The host should recognise either signal: a six-0xFF packet, or the appearance of ASCII text after a DAQ:STOP command.

14.10 SET:DIG#<n> <value> is permissive on value

Any non-zero integer value drives the pin HIGH. SET:DIG#5 2 is equivalent to SET:DIG#5 1. Non-numeric input is parsed as zero by atoi() and drives the pin LOW. The same applies to SET:DIG:ALL.

14.11 SET:PWM accepts non-PWM pins silently

The firmware does not check that the requested pin is PWM-capable. analogWrite on a non-PWM pin is a silent no-op in the Arduino core. Verify the pin choice against the Arduino reference for the board.

14.12 Pins D0 and D1 are not protected

The firmware does not refuse pinMode or digitalWrite on pins D0 and D1. Sending CONF:DIG:OUTPUT#0 or SET:DIG#1 1 will reconfigure the USB-serial pins and break further communication until the board is reset.

14.13 Command buffer overflow is silent

If more than (command_buffer_size − 1) characters arrive before a CR/LF, the excess is discarded and parsing proceeds on whatever fit. There is no [ERR] for this condition. Keep host-side lines within the buffer limit (80 bytes for Uno/Nano/Micro, 256 for Mega).

14.14 No runtime introspection for buffer sizes

There is no serial command that returns CMD_BUFFER_SIZE, SPI_BUFFER_SIZE, I2C_BUFFER_SIZE, or UART_BUFFER_SIZE. Hosts must look these up using *BOARD? and the table in §1.2.

14.15 No *OPC? or READY? query

The startup READY line is the only synchronisation signal. A host that connects after the firmware has finished booting cannot re-query readiness over the serial interface. *IDN? or any benign query (e.g. CONF:I2C:GET?) is the practical liveness check.

14.16 Reserved EEPROM regions are not yet used

The EEPROM layout reserves blocks at 0x0010–0x001F (PWM), 0x0020–0x002F (system), and 0x0030–0x003F (calibration). These are placeholders for future firmware revisions and have no effect on current behaviour.

14.17 I2C configuration is not persisted

I2C mode, speed, and address live in RAM only. After every cold boot or *RESTART, the I2C peripheral is disabled and must be reconfigured by the host. SPI is the only peripheral whose configuration is persisted in EEPROM.

14.18 DAQ timeout default is 30 minutes

The default DAQ:TIMEOUT is 1800 seconds (30 minutes), not 30 seconds. Set an explicit timeout with DAQ:TIMEOUT <seconds> if a long-running default is not what you want.

14.19 Wire library RX bytes beyond 32 may linger (Mega and Micro)

On Mega and Micro, the I2C slave RX ISR drains only the first 32 bytes per interrupt. Surplus bytes remain in the Wire library's internal buffer. On Uno and Nano, the firmware uses a drain-all pattern that keeps the Wire queue clean. With a 32-byte ring buffer this rarely matters in practice but is worth noting for high-throughput slave designs.

14.20 Master-read of I2C slave clears the sent flag on next I2C:SENT? only

I2C:SENT? returns the byte count of the most recent master-read transaction and resets the internal sent flag when it returns a non-zero count. The flag is not reset on the zero-count path. Functionally equivalent today but worth knowing if you rely on the flag's semantics.

14.21 Square wave below ~31 Hz may not be stable

Arduino's tone() library uses Timer2 on Uno/Nano and Mega by default. Below approximately 31 Hz the timer cannot produce a clean tone. SET:SQUARE:ON does not enforce this lower bound; the responsibility is on the host.

14.22 Buffer scratch for I2C:LOADTX differs across boards

Internally, the parser scratch for I2C:LOADTX is 80 characters on Uno/Nano/Micro and 128 on Mega. This bounds the maximum length of a single I2C:LOADTX command line. The TX ring itself is 32 bytes (effective payload 31 bytes) on every board.

14.23 UART output bytes are truncated silently on overlong WRITE

UART<n>:WRITE copies the parameter list into a CMD_BUFFER_SIZE scratch. If the line exceeds the scratch (80 bytes Micro, 256 bytes Mega), extra bytes are dropped before parsing. There is no [ERR] for this case. Keep UART:WRITE lines within the command buffer limit.

14.24 *RESTART output may not flush before reset

The firmware prints "Restarting...", waits 200 ms, then arms the watchdog. If the host's serial driver buffers more than 200 ms of output, late traffic from before the restart may be cut off.

14.25 Single-byte-per-loop serial intake

The main loop reads at most one byte per iteration from the host. This is deliberate so that DAQ, SPI, I2C, and UART updates run between bytes. The practical effect is that commands sent back-to-back without delays may take an extra tick to process — irrelevant for human-paced interaction, but worth knowing for tight scripted workflows.

14.26 Command parser upper-cases the entire buffer

The parser upper-cases every byte in the command line, including parameter values. This is harmless for hex digits (a→A, etc.) and for numeric values. The firmware does not accept any case-sensitive payloads at present.

14.27 Some success messages bypass DEBUG:OFF

A subset of write/configure commands prints status text unconditionally regardless of DEBUG state: "SPI Master saved. *RESTART", "SPI Slave saved. *RESTART", "SPI disabled", "I2C Buffers cleared", "DAQ stopped", "Restarting...". These are intentional, but worth knowing if you expect total silence with DEBUG:OFF.

14.28 SPI configuration loaded once at boot

The firmware reads the SPI EEPROM configuration only once, during setup(). There is no command to reload it at runtime without restarting. *RESTART is the only way to apply changes from CONF:SPI:MASTER, CONF:SPI:SLAVE, or *EEPROM:RESET.

Appendix A. Command Quick Reference

Compact list of every command, sorted by module. Boards column: U=Uno, N=Nano, m=Micro, M=Mega.

System

Command	Description	Boards
*IDN?	Identity string	U N m M
*BOARD?	Board identifier	U N m M
*RST	Reset I/O state	U N m M
*RESTART	Reset MCU via watchdog	U N m M
*EEPROM:RESET	Clear EEPROM config	U N m M
GET:ALL:PIN:MODE?	Query all pin modes	U N m M

Digital I/O

Command	Description	Boards
CONF:DIG:INPUT#<n>	Pin to INPUT	U N m M
CONF:DIG:OUTPUT#<n>	Pin to OUTPUT	U N m M
CONF:DIG:INPUT:ALL	All pins INPUT	U N m M
CONF:DIG:OUTPUT:ALL	All pins OUTPUT	U N m M
SET:DIG#<n> <v>	Write a pin	U N m M
SET:DIG:ALL <v>	Write all pins	U N m M
GET:DIG#<n>?	Read a pin	U N m M
GET:DIG:ALL?	Read all pins	U N m M
GET:DIG:MULTIPLE #<p1>,<p2>...?	Read multiple	U N m M

Analog Input

Command	Description	Boards
CONF:ANALOG:VREF [<v>]	Set/query VREF	U N m M
GET:ANALOG#<n>?	Read one channel	U N m M
GET:ANALOG:ALL?	Read all channels	U N m M
GET:ANALOG:MULTIPLE #<c1>,...?	Read multiple	U N m M

PWM and Square Wave

Command	Description	Boards
SET:PWM <pin>,<duty>	PWM 0–100 %	U N m M
SET:SQUARE:ON	Start square wave	U N m M

Command	Description	Boards
<pin>,<freq>		
SET:SQUARE:OFF <pin>	Stop square wave	U N m M

DAQ

Command	Description	Boards
DAQ:START <ch1>[,<ch2>],<f>	Begin streaming	U N m M
DAQ:STOP	Stop streaming	U N m M
DAQ:TIMEOUT [<s>]	Set/query auto-stop	U N m M
DAQ:TIMEOUT?	Query auto-stop	U N m M

SPI

Command	Description	Boards
CONF:SPI:MASTER <s>,<m>,<o>	Configure master (EEPROM)	U N m M
CONF:SPI:SLAVE <m>,<o>	Configure slave (EEPROM)	U N m M
CONF:SPI:DISABLE	Disable SPI	U N m M
CONF:SPI:GET?	Query config	U N m M
SPI:SEND <cs>,<a>,<b>,...	Send bytes (master)	U N m M
SPI:REQUEST <cs>,<a>,<n>	Read N bytes (master)	U N m M
SPI:READ?	Drain RX ring	U N m M
SPI:SENT?	Last TX count (slave)	U N m M
SPI:AVAILABLE?	RX byte count	U N m M
SPI:CLEAR	Clear buffers	U N m M

I2C

Command	Description	Boards
CONF:I2C:MASTER <s>	Configure master (100/400 kHz)	U N m M
CONF:I2C:SLAVE <a>	Configure slave (hex addr)	U N m M
CONF:I2C:DISABLE	Disable I2C	U N m M
CONF:I2C:GET?	Query config	U N m M
I2C:SCAN?	Bus scan (master)	U N m M
I2C:WRITE <a>,<b>,...	Write to slave (master)	U N m M
I2C:READ <a>,<n>	Read from slave (master)	U N m M
I2C:WRITEREAD <a>,<n>,<b>,...	Write then read (master)	U N m M
I2C:RECEIVED?	Drain slave RX ring	U N m M

Command	Description	Boards
I2C:LOADTX <b>,...	Load slave TX scratch	U N m M
I2C:SENT?	Last TX count (slave)	U N m M
I2C:LASTTX?	Last TX bytes (slave)	U N m M
I2C:AVAILABLE?	Slave RX byte count	U N m M
I2C:CLEAR	Clear buffers	U N m M

UART

Command	Description	Boards
CONF:UART<n>:BEGIN <baud>	Open channel	m M
CONF:UART<n>:END	Close channel	m M
CONF:UART<n>:GET?	Query channel state	m M
UART<n>:WRITE <b>,...	Send bytes	m M
UART<n>:READ?	Drain RX ring	m M
UART<n>:AVAILABLE?	RX byte count	m M
UART<n>:CLEAR	Clear RX ring	m M

Micro: channel 1 only. Mega: channels 1, 2, 3.

Debug

Command	Description	Boards
DEBUG:ON	Enable echoes	U N m M
DEBUG:OFF	Disable echoes (default)	U N m M

End of document.